НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено

Завідувач кафедри

_____ О.В. Коваль
(підпис) (ініціали, прізвище)

“___”_____2019р.

ДИПЛОМНА РОБОТА
на здобуття ступеня бакалавра

з напряму підготовки 6.050103 “ Програмна інженерія ”

на тему “Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++”

Виконав: студент IV курсу, групи ТІ-51

_____Скоробогатський Дмитро Володимирович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник ___доцент, к.т.н., Кузьменко Ігор Миколайович____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент___ _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший, бакалаврський

Напрям підготовки 6.050103 “ Програмна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
“___”_____2019р.

ЗАВДАННЯ
на дипломну роботу студенту

Скоробогатському Дмитру Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++”
керівник роботи Кузьменко Ігор Миколайович, к.т.н. доцент
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від 22 05 2019р. № 1325-с

2. Строк подання студентом роботи 10 червня 2019р.
3. Вихідні дані до роботи розроблений модуль написаний мовою C++ на платформі Windows.

4.Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити) розробити симуляцію фізичних об’єктів, провести аналіз існуючих методів синхронізації об’єктів в схожих системах, опрацювати матеріали і літературу відповідно до теми, спроектувати однорангову клієнт-серверну систему, обрати відповідні інструменти розробки, порівняти синхронізацію об’єктів з використанням TCP і UDP, реалізувати для системи протокол RUDP, протестувати налагоджений функціонал системи.

5. Перелік ілюстративного матеріалу: титульний аркуш, поставлені задачі, симуляція фізичних об’єктів, вибір підходу синхронізації об’єктів, порівняння протоколів TCP і UDP, структура синхронізації подій, проблеми UDP, які вирішуються в RUDP, реалізація RUDP, документація та юніт-тести, приклади стресового тестування роботи TCP і UDP, приклад стресового тестування UDP, приклад стресового тестування роботи RUDP, висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання ”_13_”_вересня_2018__р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі	14.01.2019	Вик.
2	Пошук інструментарію для розробки симуляції фізичних об’єктів	11.02.2019	Вик.
3.	Вибір підходу синхронізації об’єктів	18.02.2019	Вик.
4.	Проектування фізичної симуляції в контексті клієнт-серверної системи	25.02.2019	Вик.
4.	Розробка симуляції фізичних об’єктів. Базова синхронізація об’єктів за допомогою TCP і UDP.	18.03.2019	Вик.
	Реалізація протоколу RUDP.	16.04.2019	Вик.
	Тестування налагодженого функціоналу системи	6.05.2019	Вик.
5.	Оформлення пояснювальної записки	15.05.2019	Вик.
6.	Захист програмного продукту	18.05.2019	Вик.
7.	Передзахист	1.06.2019	Вик.
8.	Захист	19.06.2019	

Студент _____ __Скоробогатський Д. В._
(підпис) (прізвище та ініціали,)

Керівник роботи _____ ____Кузьменко І. М.___
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Дипломну роботу виконано на 82 аркушах, вона містить 4 додатки та перелік посилань на використані джерела з 7 найменувань. У роботі наведено 20 рисунків та 2 таблиці. Реалізовано синхронну поведінку у симуляції фізичних об’єктів клієнт-серверної системи, використовуючи мову програмування C++. Програма використовує розроблений гнучкий механізм передачі даних з використанням протоколу RUDP. Це дає можливість самостійно обирати, чи потрібно гарантувати доставку даних до отримувача, чи проігнорувати надійність. Система може використовуватись у симуляціях, які потребують встановлювання зв’язку між учасниками, проте використовують ненадійний протокол передачі даних.

Ключові слова: симуляція, синхронізація об’єктів, проблеми використання UDP, гарантування доставки пакетів, RUDP, Box2D, C++.

ABSTRACT

The graduate work is completed on 82 sheets, it contains 4 applications and a list of references to used sources of 7 titles. There are 20 drawings and 2 tables in the work. Synchronous behavior is implemented in physical objects’ simulation of the client-server system, using the programming language C++. The program uses the developed flexible data transfer mechanism using the RUDP protocol. This allows you to independently choose whether to guarantee the delivery of data to the recipient, or to ignore the reliability. The system can be used in simulations that require communication between participants, but use an unreliable data transfer protocol.

Keywords: simulation, object synchronization, UDP problem, packet delivery guarantee, RUDP, Box2D, C++.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OSI (Open Systems Interconnection Basic Reference Model) — абстрактна мережева модель для комунікацій і розробки мережевих протоколів.

IP (Internet Protocol) — протокол мережевого рівня для передавання датаграм між мережами.

TCP (Transmission Control Protocol) — разом із протоколом IP є стрижневим протоколом Інтернету, який дав назву моделі TCP/IP.

UDP (User Datagram Protocol) — один із протоколів в стеку TCP/IP. Від протоколу TCP він відрізняється тим, що працює без встановлення з'єднання..

RUDP (Reliable User Datagram Protocol) — мережевий протокол, що вважається надійним порівняно з UDP.

RTT (Round trip time) – час необхідний для отримання відповіді після відправки запиту.

ВСТУП

На даний час важко переоцінити розвиток та потужність клієнт-серверних додатків. Це, зокрема, і соціальні мережі, месенджери, які дозволяють обмінюватися текстовими повідомленнями, мережа Інтернет загалом, комп’ютерні ігри. Перераховані застосунки потребують певної архітектурної технології, яка дозволить передавати дані між екземплярами програми найшвидше і з мінімізацією помилок.

Архітектура “клієнт-сервер” визначає загальні принципи взаємодії між комп'ютерами, правила визначають протоколи взаємодії, наприклад: HTTP, FTP, POP, SMTP, TELNET.

В даній роботі необхідно збудувати клієнт-серверну систему для фізичної симуляції, достатньої для можливої організації синхронізації її об’єктів між декількома процесами. Система має забезпечувати детермінованість, тому фізичні характеристики предметів в динаміці (місце розміщення, швидкість руху, орієнтація) мають точно передаватися усім клієнтам системи. Необхідно роздивитися можливі варіанти синхронізації фізичних об’єктів у системі і обрати найоптимальніший в контексті поданої симуляції. Вирішення даних задач, безумовно, передбачає використання сокетів для роботи з транспортним рівнем моделі OSI.

Широко відомими протоколами транспортного рівня є TCP і UDP. Проте механізми роботи цих протоколів можуть складати певні труднощі у клієнт-серверних застосунках. Використання TCP, а саме механізм, який гарантує надійну доставку пакетів, може спричиняти затримку, що іноді є надмірною для системи. Використання UDP не створює таких затримок, проте не є надійним взагалі. Конкретні проблеми використання цих протоколів будуть описані в розділі Б.

Необхідно роздивитися реалізацію протоколу для клієнт-серверної системи, який поєднує в собі ефективність UDP і надійність TCP. Такий

протокол існує і має назву RUDP[1]. Його використовують для забезпечення надійної передачі даних між пакетно-орієнтованими застосунками. RUDP забезпечує такі функції, як підтвердження доставки пакетів, повторна відправка втрачених пакетів.

Багато існуючих клієнт-серверних застосунків мають необхідну для відправки інформацію, причому одна її частина може вимагати надійності відправки, а інша – ні. Проте використовується зазвичай один протокол у системі, що зумовлює вибір розробника у сторону ефективності чи надійності. Важливо зазначити, що клієнт розроблюваного застосунку матиме можливість самостійно вирішувати в яких випадках, йому необхідна гарантія доставки його повідомлень, а в яких цим можна знехтувати. Я впевнений, що гнучкість такого підходу, безперечно, має місце у сучасних клієнт-серверних симуляційних додатках.

1. РЕАЛІЗАЦІЯ СИНХРОННОЇ ПОВЕДІНКИ ФІЗИЧНИХ ОБ'ЄКТІВ У КЛІЄНТ-СЕРВЕРНІЙ СИСТЕМІ З ВИКОРИСТАННЯМ C++

На сьогоднішній день з величезною швидкістю збільшується кількість програмних застосунків, які є фізичними симуляціями. Зокрема, це комп'ютерні ігри, моделювання певних фізичних явищ або об'єктів реального світу та просто персональні проекти.

Зосереджуючи увагу на клієнт-серверних симуляціях, у яких за рух фізичних об'єктів відповідають користувачі, а не програмний алгоритм, слід підкреслити деякі важливі моменти. Реалізація зміни положення об'єктів у таких системах може відбуватися двома підходами: кінематичним або динамічним. Використовуючи кінематичний підхід, система розраховує положення об'єкта за певними формулами, вказаними програмістом, залежно від часу, направлення руху тощо. Динамічний підхід зумовлює рух елементів симуляції шляхом прикладання до них імпульсів і сил, що робить взаємодію об'єктів більш правдоподібною, проте, водночас, і складнішою. Складність полягає у тому, що розрахувати локально нові положення відповідно до зміни часу буде майже неможливо, так як сили мають тенденцію затухати. Отже, симуляція буде синхронізуватися шляхом передачі відповідних пакетів мережею. Таким чином, ускладнюючи процес синхронізації об'єктів у системі, можна сформулювати першу задачу – організувати симуляцію фізичних об'єктів, використовуючи динамічний підхід розрахунку положення тіл.

Існують декілька способів синхронізації фізичних об'єктів: синхронізація шляхом передачі клієнтського вводу симуляційних команд (отриманих від клавіатури або комп'ютерної миші, наприклад), синхронізація стану об'єктів (для розроблюваної системи, положення і кут об'єкту в симуляції окремо один

від одного) та інтерполяція кадрів симуляції (повна синхронізація поточного стану симуляції загалом).

Планується, що у даній системі кожен учасник симуляції буде відповідати за синхронізацію поведінки і положення власного об'єкта і сповіщати про ці зміни інших учасників. Виходячи з цього, кращим варіантом для розроблюваної системи буде синхронізувати стани об'єктів[2].

Проте незалежно від алгоритму синхронізації, система не буде захищена від таких мережевих проблем, як затримка або втрата пакетів. А тому наступна задача – гарантувати системі надійну доставку пакетів, їх порядок і, найголовніше, актуальність в рамках розроблюваної симуляції. Під актуальністю пакета я маю на увазі відсутність новіших пакетів для синхронізації об’єктів. Необхідно перевірити зазначену синхронізацію за допомогою TCP, UDP та RUDP – протокол, який має вирішити проблему надійності, актуальності пакетів.

Розв’язання цієї проблеми є важливим для розробників ігор, фізичних симуляцій та інших клієнт-серверних застосунків, тому, що іноді втрата одного пакету унеможливлює подальше використання програми.

Для впевненості в тому, що система працює коректно і виконує поставлені перед нею задачі слід розробити юніт-тести. Роботу розробленого протоколу слід протестувати за допомогою стресового тестування. Таке тестування допомагає визначити продуктивність і швидкість програми під певними навантаженнями. В контексті розроблюваної симуляції планується перевірити коректність і точність синхронізації фізичних об’єктів при симуляції втрати пакетів, їх дублювання, а також зміни порядку під час відправки.

Як результат, потрібно розробити систему, якій будуть притаманні наступні властивості:

- Інтуїтивно зрозуміла симуляція для користування;
- Синхронна поведінка фізичних об’єктів для всіх учасників симуляції;
- Гарантія доставки, порядку і коректної обробки мережевих повідомлень у випадку втрати, дублікації пакетів;
- Гнучкий підхід у виборі пакетів, доставку яких необхідно гарантувати.

2. АНАЛІЗ ПРОБЛЕМИ СИНХРОНІЗАЦІЇ ФІЗИЧНИХ ОБ'ЄКТІВ У КЛІЄНТ-СЕРВЕРНІЙ СИСТЕМІ

Було вирішено реалізовувати синхронізацію стану фізичних об'єктів у розроблюваній системі. Для початку необхідно розробити фізичну систему, яка дозволятиме створювати і керувати фізичним об'єктом за допомогою доступних засобів вводу, зокрема клавіатури і комп'ютерної миші. Слід врахувати, що у багатьох симуляціях (та іграх) оновлення об'єкта, тобто зміна його характеристик, відбувається окремо від відображення.

Наступним кроком необхідно проаналізувати, який саме протокол найкраще підходить для вирішення задачі синхронізації створених об'єктів у симуляції.

2.1 Аналіз протоколів транспортного рівня моделі OSI

Протокол транспортного рівня у моделі OSI передбачає логічний зв'язок між процесами застосунку, що виконуються на різних хостах. Під логічним зв'язком мається на увазі, що з точки зору програми, ці хости безпосередньо підключені один до одного, хоча, насправді, вони можуть бути у протилежних кутках планети, з'єднані через численні маршрутизатори і широкий спектр типів зв'язків. Інтернет, або навіть в цілому мережа TCP/IP, робить два різних протоколи транспортного рівня доступними для прикладного рівня.

Протоколи TCP і UDP відносяться до транспортного рівня моделі OSI. UDP не встановлює з'єднання, не гарантує доставку і навіть порядок пакетів. TCP – це протокол із установленням з'єднання і з гарантованою доставкою пакетів. Спочатку відбувається "рукостискання", після чого з'єднання вважається встановленим. Далі по цьому з'єднанню в обох напрямках надсилаються пакети, причому з перевіркою, чи дійшов пакет до одержувача. Якщо пакет загубився,

або дійшов, але з неправильною контрольною сумою, то він надсилається повторно. Таким чином TCP більш надійний, але він складніше з точки зору реалізації і відповідно вимагає більше пам'яті.

UDP виконує незначну роботу, що й транспортний протокол. Окрім функції мультиплексування і демультиплексування, цей протокол нічого не додає до IP. Мультиплексування – це розширення служби доставки від хоста до хоста, що надається мережним рівнем, до служби доставки від процесу до процесу для застосунків, запущених на хостах Насправді, якщо розробник програми вибирає UDP замість TCP, то додаток практично безпосередньо взаємодіє з IP. UDP бере повідомлення з процесу додатка, приєднує поля номерів вихідного і цільового портів для служби мультиплексування і демультиплексування, додає два інших невеликих поля і передає отриманий сегмент мережевому рівню. Мережевий шар інкапсулює сегмент транспортного шару в IP дейтаграму, а потім намагається доставити сегмент до приймаючого хоста. Якщо сегмент надходить на прийом хосту, UDP використовує номер порту призначення, щоб доставити дані сегмента до правильного процесу застосування. Проте під час використання UDP не має зв'язку між відправленням і прийомом об'єктів транспортного рівня перед відправкою сегмента. З цієї причини і утверджується, що UDP не встановлює з'єднання.

TCP використовує механізм тайм-ауту і повторної відправки для відновлення втрачених сегментів, яку зображено на рис. 2.1. Хоча концепція проста, виникає багато питань, коли ми реалізуємо механізм тайм-ауту і ретрансляції у схожому протоколі, такому як TCP. Можливо, найбільш очевидним питанням є довжина інтервалів тайм-ауту. Зрозуміло, що час очікування повинен перевищувати час, протягом якого буде встановлено з'єднання, RTT, тобто час того, коли сегмент надсилається, поки він не буде підтверджений. В іншому випадку, будуть відсилатися непотрібні повторні передачі.

TCP оцінює час відправлення між відправником і одержувачем наступним чином: зразок RTT для певного сегмента, позначимо DefaultRTT, - це кількість

часу між відправкою сегмента (тобто передачі в IP) і прийомом підтвердження отримання сегмента. Замість того, щоб вимірювати DefaultRTT для кожного переданого сегмента, більшість реалізацій TCP приймають тільки один розрахунок DefaultRTT за один раз. Тобто, у будь-який момент часу, DefaultRTT оцінюється тільки для одного з переданих, але в даний час не підтверджених сегментів, що призводить до нового значення DefaultRTT приблизно один раз на кожен RTT. Крім того, TCP ніколи не обчислює DefaultRTT для сегмента, який був повторно переданий.

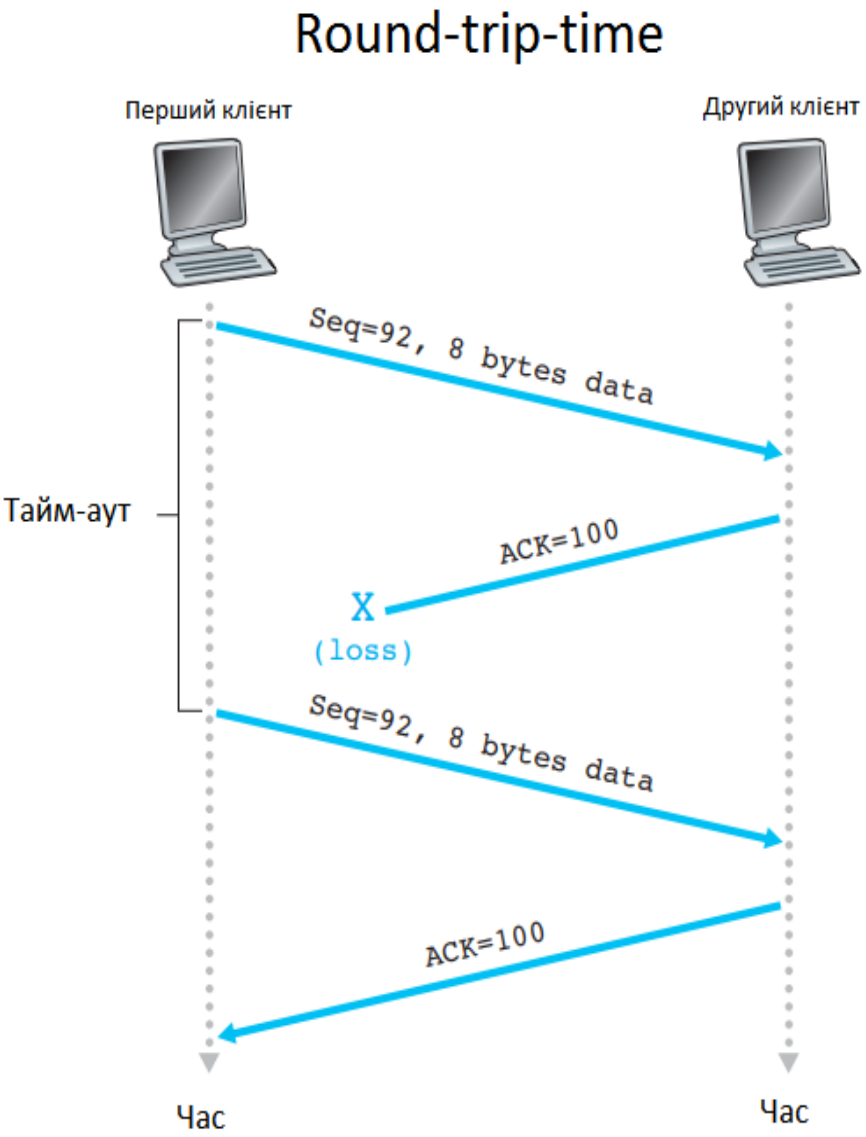


Рисунок 2.1 – Приклад передачі пакетів, використовуючи TCP

Очевидно, що значення DefaultRTT будуть коливатися від сегмента до сегмента через перевантаження в маршрутизаторах і навантаження у кінцевих системах. Для того, щоб оцінити значення RTT, достатньо прийняти деяке середнє значення DefaultRTT. TCP підтримує середнє значення, яке називається EstimatedRTT (у формулі EstRTT), із значень DefaultRTT. Після отримання нового DefaultRTT, TCP оновляє EstimatedRTT за такою формулою:

$$EstRTT = (1 - \alpha) \cdot EstRTT + \alpha \cdot DefaultRTT \quad (2.1)$$

Наведена вище формула написана у вигляді викладу мови програмування нове значення EstimatedRTT - це зважена комбінація попереднього значення EstimatedRTT і нове значення для DefaultRTT. Рекомендоване значення — $\alpha = 0{,}125$.

На додаток до оцінки RTT, також важливо мати міру мінливості RTT. Існують реалізації, які визначають значення варіації RTT, DevRTT, як оцінку того, скільки DefaultRTT зазвичай відхиляється від EstimatedRTT:

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot |DefaultRTT - EstRTT| \quad (2.2)$$

Очевидно, що інтервал повинен бути більшим або рівним EstimatedRTT, в протилежному випадку будуть відсилатися надмірні повторні передачі. Але інтервал очікування не повинен бути набагато перевищувати EstimatedRTT; інакше, коли сегмент втрачається, TCP не зможе швидко повторно передати сегмент, що призведе до великих затримок при передачі даних. Тому бажано встановити тайм-аут, рівний EstimatedRTT плюс деякий запас. Таким чином, тут використовується значення DevRTT. Всі ці міркування враховуються в методі TCP для визначення інтервалу тайм-ауту повторної передачі:

$$TimeoutInterval = EstRTT + 4 \cdot DevRTT \quad (2.3)$$

Рекомендується початкове значення часу очікування 1 секунда. Крім того, при виникненні тайм-ауту, значення TimeoutInterval подвоюється, щоб уникнути передчасного таймауту для наступного сегмента, який незабаром буде визнано. Однак, як тільки сегмент буде отримано і значення EstimatedRTT оновлено, TimeoutInterval розрахується знову, використовуючи формулу 2.3.

Протоколи транспортного рівня широко використовуються у нашому повсякденому житті. У таблиці 2.1 вказано основні застосунки, з якими нам приходиться працювати і протоколи, які для цього використовуються, для того, щоб оцінити потреби у даній симуляції.

Таблиця 2.1 – Використання протоколів транспортного рівня моделі OSI

Застосунок	Протокол прикладного рівня	Основний транспортний протокол
Електронна пошта	SMTP	TCP
Віддалений доступ до терміналу	Telnet	TCP
Веб	HTTP	TCP
Передача файлів	FTP	TCP
Віддалений сервер файлів	NFS	Зазвичай UDP
Потокове мультимедіа	Залежить від розробника	UDP або TCP
Інтернет-телефонія	Залежить від розробника	UDP або TCP
Мережеве управління	SNMP	Зазвичай UDP
Протокол маршрутизації	RIP	Зазвичай UDP
Переклад імені	DNS	Зазвичай UDP

Розроблювана система передбачає оновлення кадрів шістдесят разів у секунду. Не беручи до уваги затримку, що зумовлена передачею мережею, а, розглядаючи лише потенційні втрати пакетів при передачі, експериментально можна встановити, що втрата одного пакета створює суттєву затримку, і пакет “застаріває”. Симуляція з використанням TCP втрачатиме плавність навіть при двох відсотках втрат пакетів за рахунок механізму повторної передачі. Звичайно, у системі існують повідомлення, які ми повинні отримати, не дивлячись на затримку, проте основна синхронізація фізичних характеристик, яка відбувається кожну одну шістдесяту секунди, може допускати до десяти відсотків втрат. Експериментально було доведено, що симуляція при використанні UDP і втратах п’ятдесят відсотків є більш плавною, ніж TCP при двох відсотках втрат. А тому, власне для синхронізації фізичних об’єктів механізм роботи UDP влаштовує мої потреби.

Тому необхідно розв’язати поставлену задачу шляхом створення RUDP – протоколу, в основі якого лежить UDP, проте гарантує доставку пакетів, вирішує проблему дублікації та зміни вихідного порядку пакетів при відправці.

2.2 Аналіз існуючих реалізацій RUDP

Для реалізації RUDP було знайдено відповідну специфікацію цього протоколу. Загалом, слід зазначити, що результатом моєї роботи буде не програмний продукт, а технологія, принцип якої можна буде інтегрувати в іншу систему для забезпечення надійної доставки пакетів, вирішення питань, пов’язаних з мережевими недоліками.

Проте, все таки, можливо знайти деякі персональні реалізації цього протоколу в мережі Інтернет, зокрема, в одному з найбільших веб-сервісів для спільної розробки програмного забезпечення GitHub. Мені вдалося знайти на цьому сервісі, побудованому на системі контроля версій Git, реалізації на таких мовах програмування, як C, C++, Java, C#, Python, Go та інших. Проте я не знайшов реалізації на C++, використовуючи можливості Qt Framework. В

додаток до цього, я організував систему зв’язку клієнтів peer-to-peer (“кожний з кожним”) і на прикладі фізичної симуляції зможу продемонструвати роботу протоколу RUDP.

Проблеми протоколу UDP, які вирішує RUDP детальніше описані в підрозділі 2.3.

2.3 Аналіз проблеми збереження порядку пакетів в RUDP

Всі методи реалізації RUDP, які будуть описані в пункті 2.4 неявно залежать від доставлених дейтаграм, що мають чітко визначену семантику впорядкованої доставки. Іншими словами, якщо відправник посилає пакет A, за яким йде пакет B, то одержувач отримає A, а потім отримує B (припускаючи, що обидва пакети дійсно надходять). Оскільки UDP сам по собі не гарантує впорядковану доставку, основний крок у забезпеченні надійної доставки полягає в тому, щоб гарантувати, що пакети надходять у правильному порядку. Мова йде про UDP тому, що цей протокол буде взято за основу для реалізації RUDP.

Найпростіший підхід для збереження порядку дейтаграм полягає в призначенні кожній з них монотонно зростаючого порядкового номера [3].

Кожен клієнтський застосунок зберігає вхідний порядковий номер ***in*** і вихідний порядковий номер ***out***. ***Out*** збільшується кожен раз, коли пакет передається. Коли пакет приймається, якщо ***in*** менше або дорівнює порядковому номеру пакета ***p***, то ***in*** встановлюється в ***p*** + 1 і пакет доставляється до програми. Якщо ***in*** виявився більшим, ніж ***p***, то це майже напевно вказує на те, що пакети були доставлені не в порядку, пакет був втрачений, або пакет був дубльований, і його можна просто відкинути.

На рис 2.2 схематично вказано робочий потік без жодних проблем. Пакет 0 був надісланий і прибув. Порядковий номер пакета дорівнює порядковому номеру вхідної послідовності. Порядковий номер ***in*** у клієнта збільшується. Пакет 0 обробляється програмою. Аналогічна поведінка з пакетом 1 і так далі.

На рис 2.3 зображено ситуацію, коли ми втрачаємо пакет. Алгоритм залишається той же і коректно вирішує дану проблему. Пакети 0, 1 і 2 були відправлені. Під час процесу відправки пакет 1 руйнується через деяку помилку в мережі. Пакет 0 надходить і приймається правилами нашої послідовності. Пакет 2 надходить і також приймається. Слід звернути увагу на те, що вхідний порядковий номер на клієнтській стороні в кінці рівний трьом (next-in: 0). Це означає, що очікується пакет більший або рівний трьом.

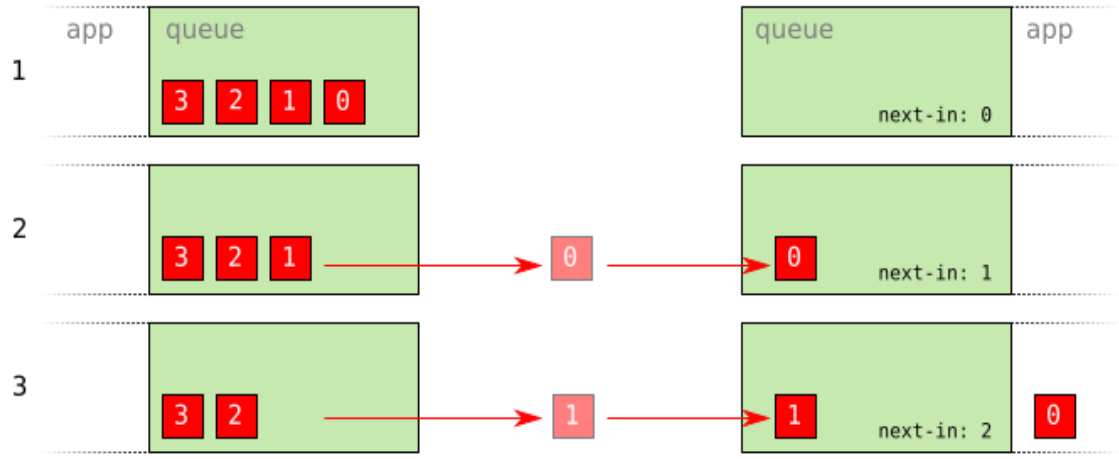


Рисунок 2.2 – Передача пакетів без мережевих проблем

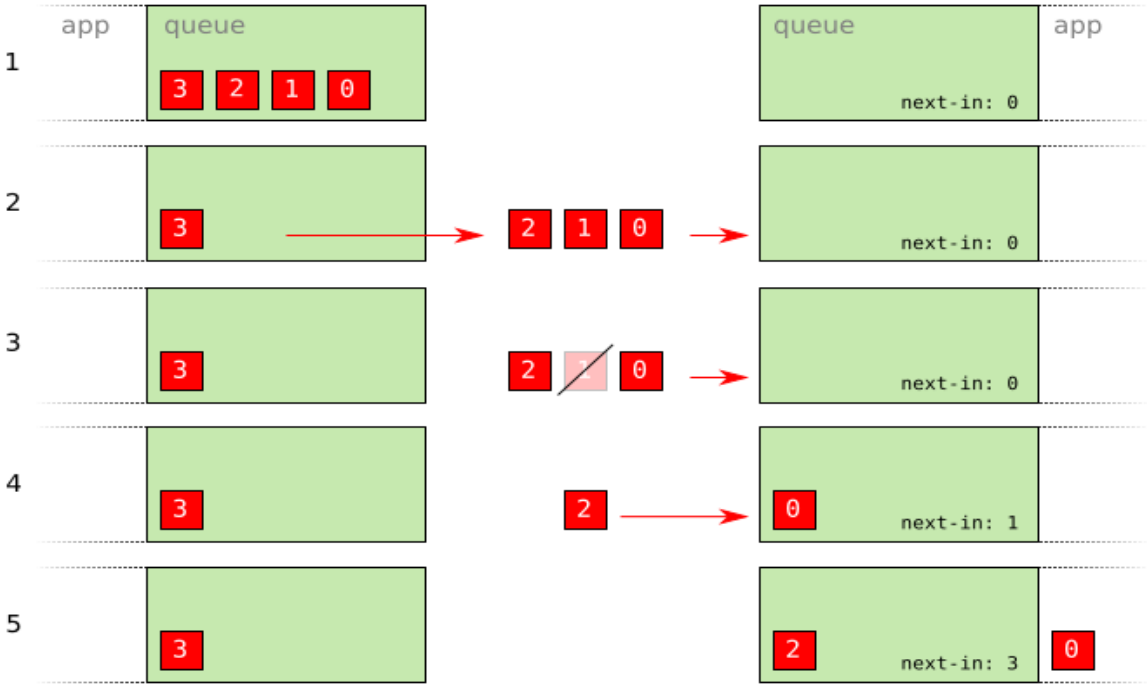


Рисунок 2.3 – Передача пакетів з можливою втратою

На рис 2.4 вказано, що також вирішується проблема дублікації пакетів. Пакет 0 відправлено. Пакет 0 дублюється через помилку в мережі. Перший екземпляр пакета 0 надходить і приймається за правилами послідовності. Приходить другий екземпляр пакета 0, але вхідний порядковий номер у клієнта тепер дорівнює одиниці. Пакет відкидається, а вхідний порядковий номер залишається незмінним.

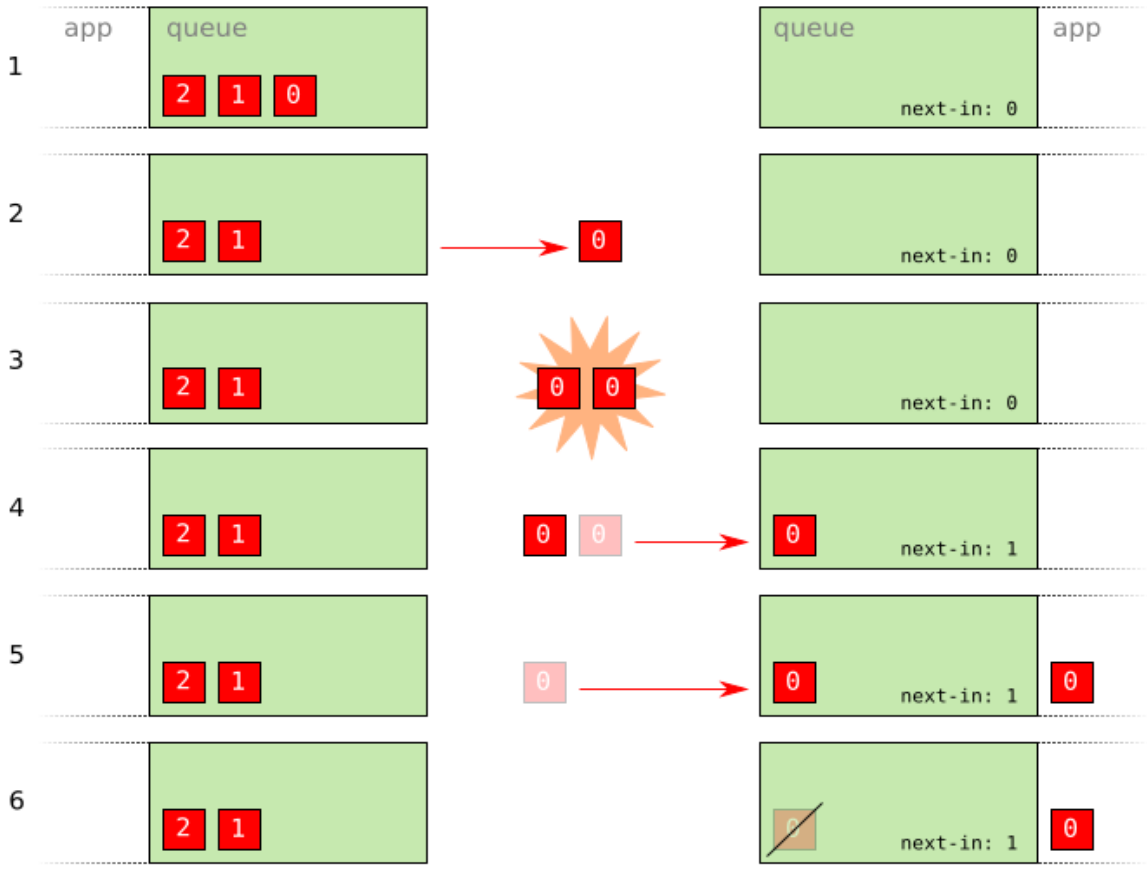


Рисунок 2.4 – Передача пакетів з можливою дублікацією

І останню можливу ситуацію зображено на рис 2.5, в якій порушується порядок відправлених пакетів. Пакети 0 і 1 були відправлені. Вони переупорядковуються під час відправки через помилку в мережі. Пакет 1 надходить і приймається за правилами послідовності. Пакет 0 надходить і відкидається, бо вхідний порядковий номер дорівнює двом.

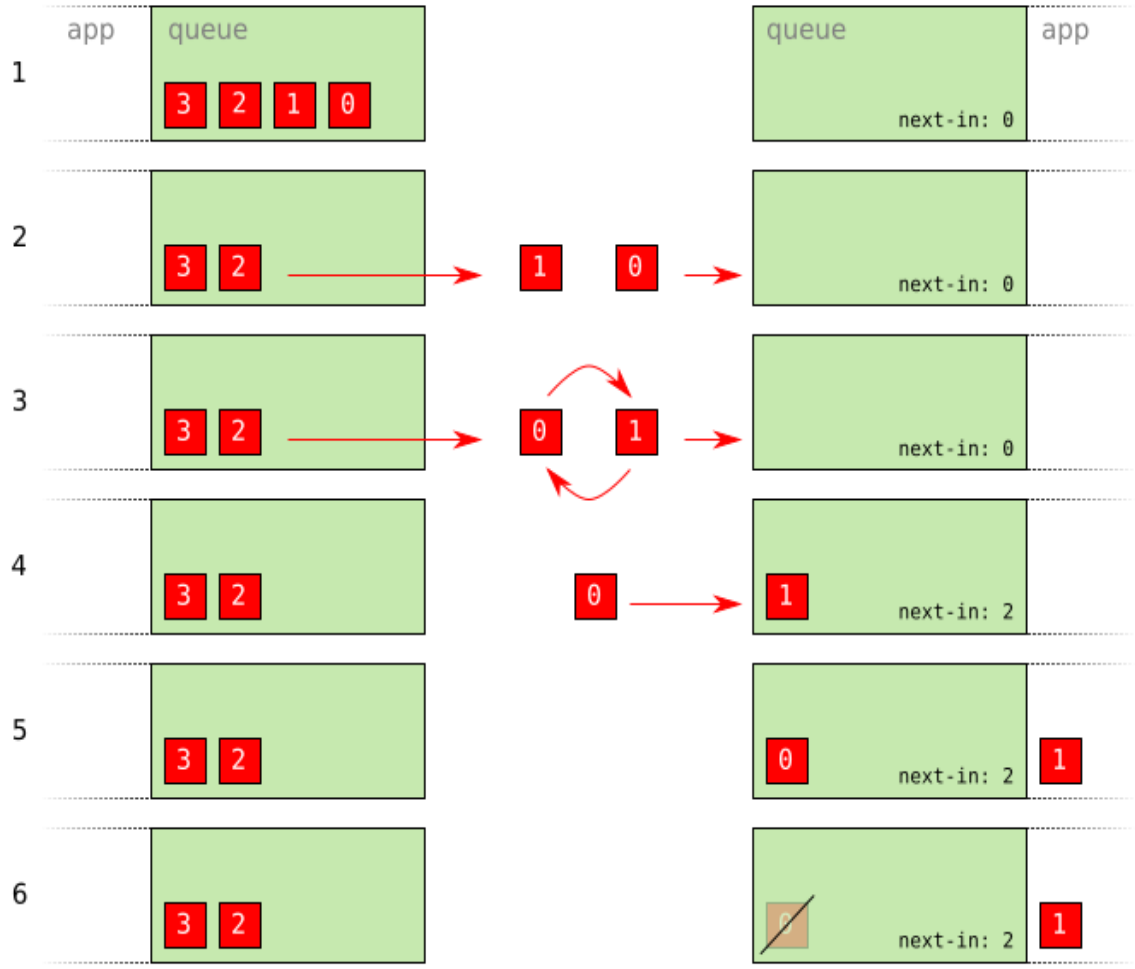


Рисунок 2.5 – Передача пакетів з можливим порушенням порядку

2.4 Аналіз існуючих методів реалізації надійної доставки пакетів в RUDP

Перший з алгоритмів і найпростіший водночас забезпечення надійної доставки вимагає підтвердження для спеціально позначених пакетів. Якщо пакет ***p*** позначений як надійний, відправник не буде відправляти будь-які подальші пакети, поки приймач не відповів пакетом підтвердження, який посилається на порядковий номер ***p***. Якщо відправник не отримав підтвердження для ***p*** протягом налаштованого терміну, відправник повторно посилає ***p***. Як правило, максимальне число повторних посилань є конфігурованим значенням. Коли максимальна кількість повторних посилань була досягнута без підтвердження, приймач вважається недосяжним і програма повідомляється про це. Часто

реалізація використовує експоненціальне відхилення у часі повторних посилань, припускаючи, що пакети втрачаються через перевантаження мережі, і тому якщо передавати пакети рідше, перевантаження зменшиться.

Проте слід зазначити, що повідомлення, які потребують і не потребують надійної доставки знаходяться в одній черзі. На рис 2.6 зображено робочий потік описаного методу. Пояснити рисунок можна так: відправляється надійний пакет 0, клієнт отримує пакет 0. Пакет 0 знаходиться в черзі на стороні отримувача, формується підтвердження для цього пакета і надсилається до відправника. Приходить підтвердження для 0, і наступний пакет може бути надісланий.

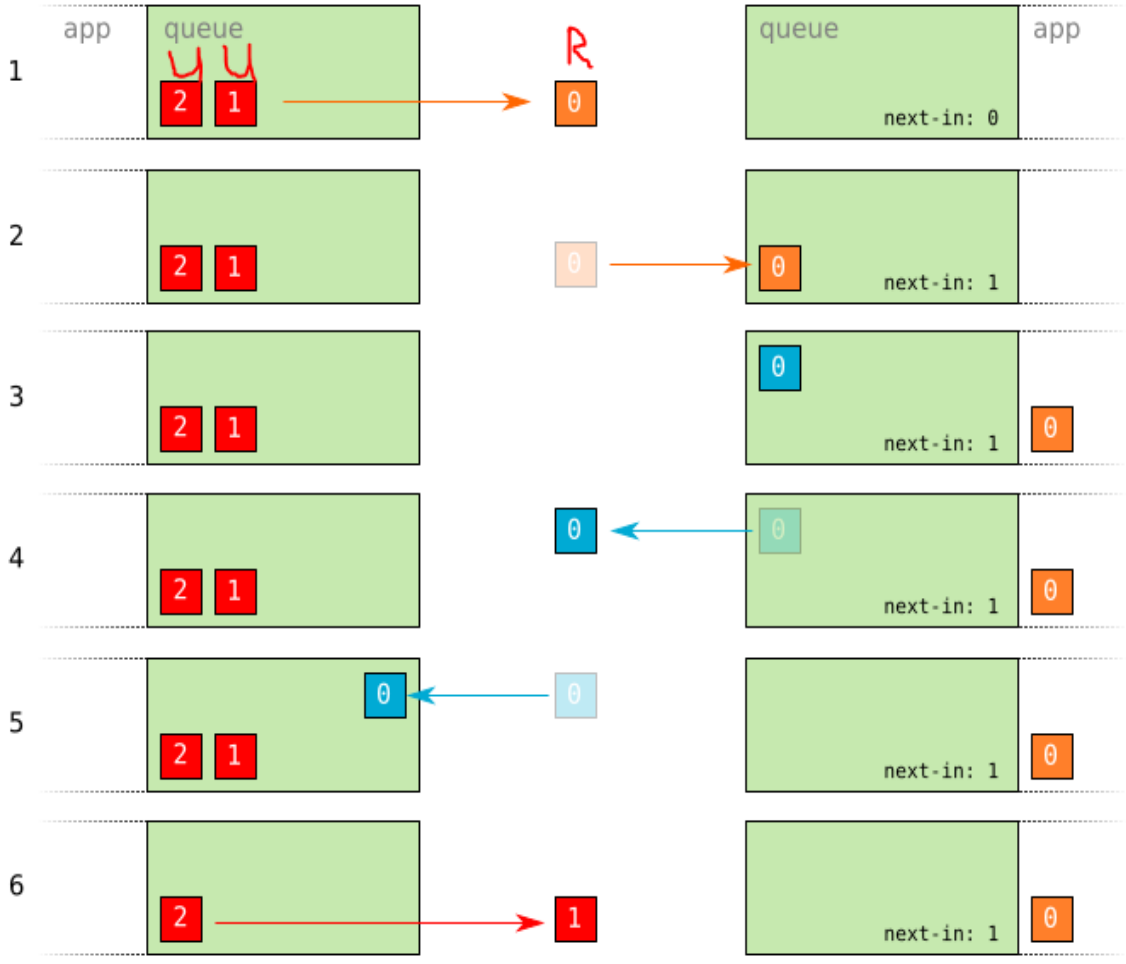


Рисунок 2.6 – Найпростіший метод гарантування доставки пакетів

Хоча цей підхід дуже простий, він має суттєві проблеми. По-перше, основною метою всієї системи може бути передача критично важливих даних. У

випадку втрати надійного пакета або підтвердження, відправлення інших пакетів припиняється, поки не буде доставлений надійний пакет. По-друге, система припускає, що в будь-який момент часу відправлятися буде не більше одного надійного пакета. Це фактично означає, що в кращому випадку швидкість передачі для надійних пакетів зводиться до значення, рівного часу туру в обидва кінці (round-trip-time, RTT). Це серйозна проблема для додатків, яким необхідно надсилати і отримувати потік пакетів з дуже високою швидкістю.

Для покращення цього алгоритму необхідно розділити чергу на дві: одну для “ненадійних” пакетів, іншу – для “надійних”, відповідно додавши відповідні вхідні та вихідні порядкові номера для кожної групи. Наступним кроком оптимізації даного підходу буде незалежна відправка кожного повідомлення незалежно від типу, щоб уникати очікування доставки попереднього повідомлення. Проте є нюанс, який необхідно зазначити: у такому випадку, необхідно буде буферизувати отримані повідомлення для отримання коректної за порядком послідовності. Конкретніше цей процес відображено на рис 2.7. Зокрема, розділяючи пакети на їхні власні черги, приймач, який керує кожною чергою, має більше інформації для роботи, коли він помічає розрив у порядкових номерах – отриманого пакету і очікуваного значення ***in***. Якщо існує розрив у порядкових номерах “ненадійної” черги пакетів, його можна просто ігнорувати: пакет або втрачений, або якщо він був змінений в польоті і незабаром з'явиться, його можна безпечно відкинути, як і будь-який пізній “ненадійний” пакет. Якщо існує розрив у порядкових номерах “надійної” черги, одержувач може припустити, що відсутній пакет скоро буде відправлений відправником, і може затримати доставку будь-яких вже отриманих “надійних” пакетів. Обраний метод взято за основу реалізації RUDP в розроблюваній системі.

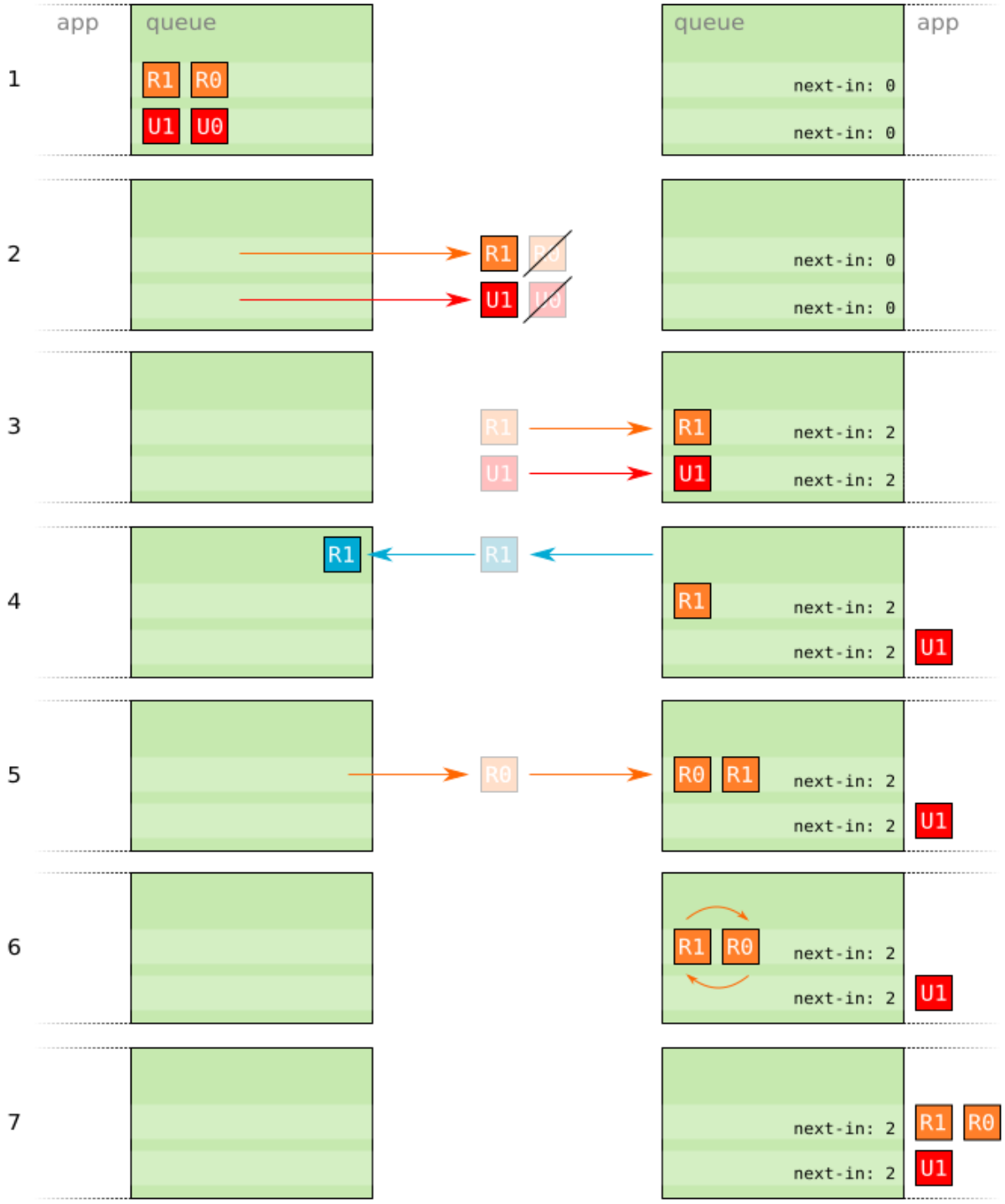


Рисунок 2.7 – Модифікований метод гарантування доставки пакетів

3. ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ СИСТЕМИ

Аналізуючи поставлену задачу та методи її вирішення, а також цільову аудиторію розроблюваної системи, було прийнято рішення розробляти систему з використанням C++ у зв'язці з Qt Framework. Qt Creator є крос-платформним засобом розробки, який став популярнішим за останні 5 років, тому я впевнений, що моя реалізація лише доповнить список вже існуючих репозиторіїв, пов'язаних з RUDP.

Реалізовані на C++[4] програмні продукти, зазвичай, відрізняються більш високою продуктивністю. Хоча робота з пам'яттю програмного застосунку вимагає від розробника певних знань і уваги, проте чітко спроектована система поводить себе так, як очікується (наприклад, вивільнення пам'яті в порівнянні з мовами, які мають збирач сміття).

Засіб розробки Qt Creator дозволяє досить швидко створювати програмний інтерфейс, що дає змогу швидко робити прототипи і навіть готовий інтерфейс.

3.1 Вибір засобів реалізації симуляції

Для реалізації фізичної симуляції було використано бібліотеку Box2D. Бібліотеку розроблено для моделювання твердого тіла у двовимірному просторі в іграх. Проте програмісти використовують його в своїх симуляціях також.

Бібліотека Box2D працює з числами з плаваючою точкою, і для її коректної роботи потрібно дотримуватися певних вимог: усі відстані у системі повинні задаватися в метрах, вага - в кілограмах, час - в секундах. Зокрема, цей фізичний двигун було налаштовано на роботу з переміщенням фігур розміром від 10 сантиметрів до 10 метрів, а статичні об'єкти можуть бути довжиною до 50 метрів, так як вони є нерухомими.

Використовуючи двомірний фізичний двигун, спокусливо використовувати пікселі як одиниці вимірювання, проте, це значно погіршує

симуляцію. Довжину в 200 пікселів в Box2D можна порівняти з висотою сорокап’ятиповерхового будинку.

Ефективне управління пам'яттю відіграє провідну роль у розробці Box2D. Тому при створенні тіл b2Body або з’єднань b2Joint, вам потрібно викликати фабричні функції b2World. Ви ніколи не повинні намагатися створювати ці об’єкти по-іншому.

Коли створюється тіло або з’єднання, потрібно надати їх визначення. Ці визначення містять всю інформацію, необхідну для побудови цих об’єктів. Використовуючи цей підхід, ми можемо запобігти появу помилки, тримати число параметрів функції малим, забезпечувати коректні значення за замовчуванням і зменшувати кількість методів доступу.

Фабрики об’єктів не зберігають посилання на визначення. Таким чином, можна створювати визначення на стеці і зберігати їх у тимчасових ресурсах.

Велика кількість рішень щодо проектування Box2D базувалася на необхідності швидкого і ефективне використання пам'яті. Box2D прагне виділити велику кількість дрібних об'єктів (близько 50-300 байт). Використання кучі для системи через malloc або new для малих об'єктів є неефективним і може викликати фрагментацію. Багато невеликих об'єктів можуть мати короткий час існування, наприклад, з’єднання, але можуть зберігатися протягом декількох ітерацій. Тому для Box2D реалізовано алокатор, який може ефективно забезпечувати пам'ять у кучі для цих об'єктів. Алокатор - спеціалізований клас, що реалізує та інкапсулює деталі розподілення ресурсів комп’ютерної пам’яті.

Рішення Box2D полягає у використанні алокатора малих об'єктів (SOA) під назвою b2BlockAllocator. SOA зберігає кількість зростаючих в обсязі пулів різного розміру. Коли робиться запит на виділення пам'яті, SOA повертає блок пам'яті, яка найкраще відповідає бажаному розміру. Коли блок звільняється, він повертається в пул. Обидві ці операції є швидкими.

Box2D містить простий невеликий векторний і матричний модуль. Це було розроблено з урахуванням внутрішньої потреби Box2D і API. Всі члени є публічними методами або вільними функціями, тому ви можете вільно

користуватися ними в своїй програмі. Математична бібліотека залишається простою, щоб зробити цей фізичний двигун легким для підтримки та портування на іншу платформу.

Частиною бібліотеки є модуль Зіткнення, який містить форми та функції, які працюють з ними. Модуль також містить динамічне дерево і широка фаза для прискорення колізійної обробки у великих системах. Модуль Зіткнення призначений для використання за межами динамічної системи. Наприклад, можна використовувати динамічне дерево для інших аспектів вашої гри, крім фізичної частини. Однак основна мета Box2D полягає в тому, щоб забезпечити роботу фізичного двигуна твердих тіл, таким чином використання модуля окремо може бути обмеженим для деяких можливостей.

Об’єкти (інакше кажучи, тіла в Box2D) бувають статичні, кінематичні і динамічні. Статичне тіло не рухається у симуляції і поводить себе, наче має нескінченну масу. Внутрішньо, Box2D зберігає її у вигляді нуля. Статичні тіла можуть бути переміщені користувачем вручну. Статичне тіло має нульову швидкість. Статичні тіла не взаємодіють (зіштовхуються) з іншими статичними або кінематичними тілами.

Кінематичне тіло рухається у симуляції відповідно до своєї швидкості. На кінематичні тіла сили не впливають. Вони можуть бути переміщені вручну користувачем, але, зазвичай, рухаються шляхом встановлення їхньої швидкості. Кінематичне тіло поводиться так, ніби має нескінченну масу. Кінематичні тіла не взаємодіють (зіштовхуються) з іншими кінематичними або статичними тілами.

Динамічне тіло досить точно моделює реальний аналог. Такі тіла можуть бути переміщені користувачем вручну, але, частіше, вони рухаються внаслідок прикладання до них сил. Динамічне тіло може зіткнутися з усіма типами тіл. Динамічне тіло завжди має ненульову масу. Якщо спробувати встановити масу динамічного тіла в нуль, вона автоматично набуде значення в один кілограм, і тіло не буде обертатися.

3.2 Вибір архітектури програмного застосунку

Архітектурною основою[5] програмного застосунку було вирішено обрати клієнт-сервер. Така архітектура описує розподілені системи, що складаються з окремих клієнта і сервера і з'єднує їх в мережі. Найпростіша форма системи клієнт-сервер, що називається дворівневою архітектурою – це серверний додаток, до якого безпосередньо звертаються безліч клієнтів.

Історично ця архітектура являє собою додаток з графічним інтерфейс, обмінюватися даними з сервером бази даних, на якому у формі процедур розташовується основна частина бізнес-логіки, або з виділеним файловим сервером. Якщо розглядати більш узагальнено, архітектурний стиль клієнт-сервер описує відносини між клієнтом і одним або більше серверами, де клієнт ініціює один або більше запитів, очікує відповіді і обробляє їх при отриманні. Зазвичай сервер авторизує користувача і потім проводить обробку, необхідну для отримання результату. Для зв'язку з клієнтом сервер може використовувати широкий діапазон протоколів і форматів даних.

До інших різновидів стилю клієнт-сервер відносяться: системи клієнт-черга-клієнт, однорангові системи, сервери додатків. Було обрано однорангову (іншими словами, Peer-to-Peer, P2P) реалізацію системи. Створений на базі клієнт-черга-клієнт, стиль P2P дозволяє клієнту і серверу обмінюватися ролями з метою розподілу і синхронізації даних між безліччю клієнтів. Ця схема розширює стиль клієнт-сервер, додаючи множинні відповіді на запити, спільно використовувані дані, виявлення ресурсів і стійкість при видаленні учасників мережі.

На використання в системі клієнт-серверного стилю розробки наштовхнули безпосередньо його переваги:

- Надійність, так як всі необхідні дані зберігаються на сервері, який зазвичай забезпечує більший контроль безпеки, ніж клієнтські комп'ютери.

- Централізований доступ до даних. Оскільки дані зберігаються лише на сервері, адміністрування доступу до даних набагато простіше, ніж в будь-яких інших архітектурних стилях.
- Простота обслуговування. Клієнт гарантовано залишається необізнаним в подіях, що відбуваються з сервером.

3.3 Опис архітектури сервера і клієнта

В силу того, що розроблювана система має бути одноранговою, архітектура сервера і клієнта майже не відрізняються, окрім імітації «налаштування» сервера і клієнтського підключення до сервера. Я впровадив систему внутрішніх повідомлень, а тому ключова різниця між клієнтом і сервером – це обробка різних за типом повідомлень.

Одноранговi системи (P2P) є більш гнучкими, і, у подальшому, у разі якщо навіть сервер (або авторитарний клієнт, по-іншому) фатально припинить свою роботу, один з клієнтів системи зможе його замінити, взяти його обов’язки на себе. Це не буде так проблематично, якщо порівнювати такий вид системи з системою з виділеним сервером, де повністю всі розрахунки відбуваються на серверній стороні.

Можливий робочий потік взаємодії клієнтів з сервером можна описати наступним чином: сервер відкриває порт для прослуховування, клієнт відсилає запит на підключення до серверу, і отримує від нього відповідь. Відповідь несе в собі інформацію про інших учасників системи. Інші вже підключені до симуляції клієнти отримують інформацію про нового клієнта. Один з учасників системи, клієнт або сервер, може створити власний об’єкт, за управління і синхронізацію якого відповідає лише він. Учасник є мастером цього об’єкта. Створення об’єкта, його фізичні характеристики і видалення синхронізуються мастером між всіма іншими учасниками. Якщо клієнт припиняє своє користування системою, сервер буде сповіщено про це, і він синхронізує цю дію між іншими учасниками.

3.4 Опис інструментів розробки

Розроблювана система розроблювалася за допомогою мови програмування C++ у зв’язці з Qt Framework. Qt – крос-платформний інструментарій розробки програмного забезпечення (ПЗ) мовою програмування C++. Дозволяє запускати написане за його допомогою ПЗ на більшості сучасних операційних систем, просто компілюючи текст програми для кожної операційної системи без зміни коду.

Було обрано стандарт мови C++11, тому, що він, порівняно з C++98 і C++03, має значно потужніший функціонал. Серед його особливостей хотілося б зазначити ті, які безпосередньо використовуються в поданій системі. Такими є: ключове слово auto, що дозволяє не вказувати тип змінної явно, говорячи компілятору, щоб він сам визначив фактичний тип змінної; строго-типізований enum (тип перерахування), інтелектуальні показчики, які дозволяють безпечно виділяти пам’ять, не хвилюючись про її вивільнення; лямбда-функції.

Фреймворк Qt має механізм сигналів і слотів[6], який нагадує систему подій. Слоти з’єднуються з сигналами і викликаються, коли випускається сигнал.

Робота з сокетами відбувається через клас QUdpSocket[7]. Найпоширенішим способом використання цього класу є прив'язка до адреси та порту за допомогою методу bind(), потім виклику writeDatagram() і readDatagram() або receiveDatagram() для передачі даних.

Сокет видає сигнал bytesWritten() кожного разу, коли в мережу записується дейтаграма. Для відправлення дейтаграми не є обов’язковим виклик bind().

Сигнал readyRead() випускається кожного разу, коли надходять дейтаграми. У цьому випадку hasPendingDatagrams() повертає true. Я використовую pendingDatagramSize(), щоб отримати розмір дейтаграми, що очікується, і readDatagram() або receiveDatagram(), щоб прочитати її.

Для реалізації фізичної симуляції я використовую Box2D. Box2D - це бібліотека для моделювання твердого тіла у двовимірному просторі для ігор. Програмісти можуть використовувати його в своїх симуляціях також. Об'єкти

рухаються реалістично і роблять симуляцію більш інтерактивною. Box2D написано на C++.

Для відображення об’єктів у вікні Qt застосунку я використовую клас QGraphicsScene. QGraphicsScene забезпечує поле для керування великої кількістю двовимірних графічних елементів. Клас служить контейнером для QGraphicsItems. Власне QGraphicsItems я поєдную з фізичними тілами з Box2D. Він використовується разом з QGraphicsView для візуалізації графічних елементів, таких як прямокутники, текст та інші спеціальні елементи на площині.

Клас QGraphicsScene також надає функціональні можливості, які дозволяють ефективно визначати, як розташовуються елементи, чи є елементи видимими в межах довільної області на сцені. За допомогою віджета QGraphicsView можна візуалізувати всю сцену або збільшити масштаб і переглянути лише її частини.

Для налагодження системи вводу було використано метод GetAsyncKeyState(int) з заголовочного файлу windows.h для асинхронної перевірки, чи була натиснена певна клавіша.

Для обробки параметрів командного рядка я використовую клас QCommandLineParser. Він дозволяє у зручному режимі задати необхідні користувацькі параметри для командного рядка і опрацьовувати їх при запуску застосунку. Клас дозволяє перевіряти, чи були встановлені вказані аргументи і безпечно доступатися до їх значення. В розроблюваній системі я додав можливість, використовуючи параметри командного рядка, передавати позицію симуляційного вікна та номер порту серверу, який буде перезаписано. В такому випадку, якщо порт за замовчуванням зайнятий, користувач матиме змогу використовувати інший порт.

4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Система синхронної поведінки фізичних об’єктів буде складатися з чотирьох внутрішніх модулів і одного зовнішнього. Головним модулем є головне вікно застосунку(MainWindow), який водночас є елементом компонування системи і графічного інтерфейсу. Зовнішнім модулем є Box2D, що використовується для реалізації руху та розрахунку колізій фізичних об’єктів. Модуль Симуляція(Simulation) інкапсулює в собі взаємодію з Box2D. Для забезпечення вводу користувача передбачено Шар вводу(InputLayer) – модуль, який відповідає за поточний стан натиснення клавіш, не вдаючись у подробиці WinAPI реалізації. За роботу з сокетами, передачу та отримання пакетів і синхронізацію даних відповідає модуль Мережа(Network). На рис. 4.1 наведена схема структури системи, на якій розташовані всі програмні модулі. Слід додати, що до Simulation складається з симуляції (клас Simulation), симуляційного об’єкту (клас SimulatedObject) і симуляційного світу(клас SimulationWorld), який, зокрема, і відповідає за створення світу, використовуючи Box2D. Модуль Network об’єднує клас NetworkAPI (взаємодія з сокетом) і відповідні для клієнта і сервера реалізації протоколів TCP, UDP.

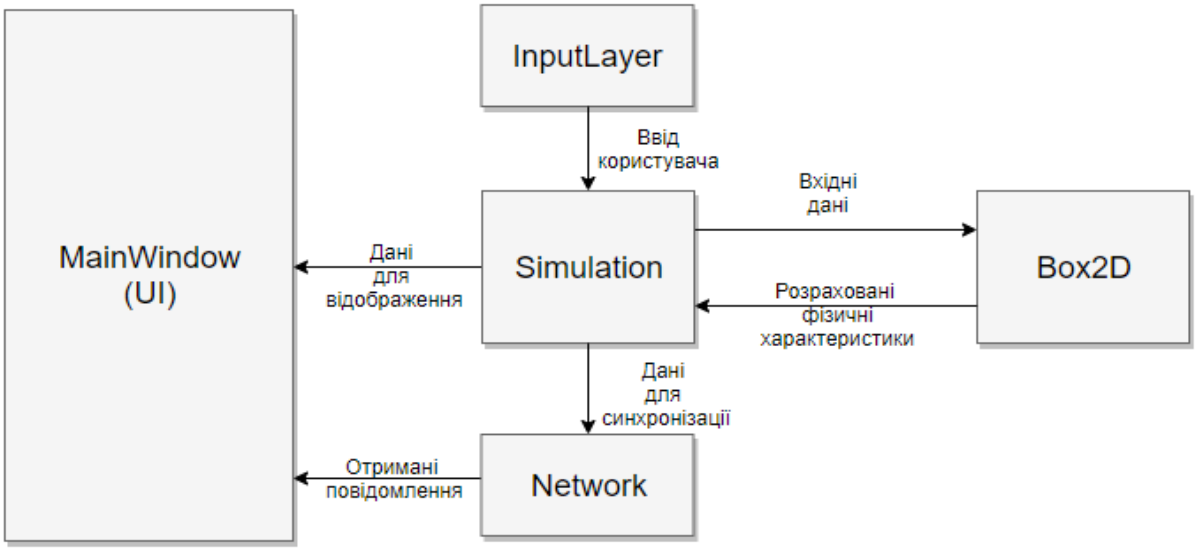


Рисунок 4.1 — Схема структури системи

4.1 Опис підсистеми “Симуляція” розроблюваної симуляції

Підсистему Симуляція(Simulation) було спроектовано таким чином, щоб вона оновлювалася 60 разів у секунду. Цю задачу було реалізовано за допомогою вбудованого у фреймворк Qt механізму сигналів і слотів. Було створено таймер всередині об’єкту Simulation, який після старту симуляції, кожну 1/60 секунди відсилає сигнал у симуляцію. Цей сигнал оброблюється слотом runFrame(). Цей метод містить у собі інші приватні методи, за допомогою яких процес оновлення симуляції поділяється на конкретні логічні частини: preUpdate(), update(), render(), postUpdate().

Метод preUpdate() відповідає за усю ініціалізацію перед безпосереднім оновленням стану симуляції, зокрема встановленням шару вводу користувача. Отримана інформація про натиснені клавіші буде готова для використання при виклику методу update().

Метод update() викликає однойменний метод усіх створених об’єктів. Вони в свою чергу, використовуючи дані про користувацький ввід с зовнішніх пристроїв, в залежності, чи були натиснені відповідні клавіші, виконують певні операції (створення, рух, синхронізація).

Метод render() також викликає однойменний метод усіх створених об’єктів. При цьому метод відображає об’єкти на симуляційному полі, використовуючи їх характеристики. Цей метод нічого не розраховує додатково.

Метод postUpdate() відновлює поточний кадр симуляції до вихідного стану, в тому числі скидання стану шару вводу.

У загальному вигляді, роботу симуляції можна описати наступним чином:

- Користувач створює симуляцію;
- Користувач створює об’єкт;
- Симуляція оновлюється кожну 1/60 секунди:
    - Система готується до оновлення стану;
    - Усі об’єкти оновлюють свій поточний стан;
    - Усі об’єкти відображаються на симуляційному полі;

- Система очищує поточний стан, готуючись до наступного кадру;

- Симуляція завершує симуляцію, усі ресурси звільняються.

Слід зазначити, що симуляція буде оновлюватися навіть, якщо користувач не створить локально об’єкт.

4.2 Опис функціональності системи

Розроблювана система містить у собі одного головного актора – користувач симуляції. На рисунку 4.2 представлена діаграма прецедентів, яка описує функції та дії актора у системі.

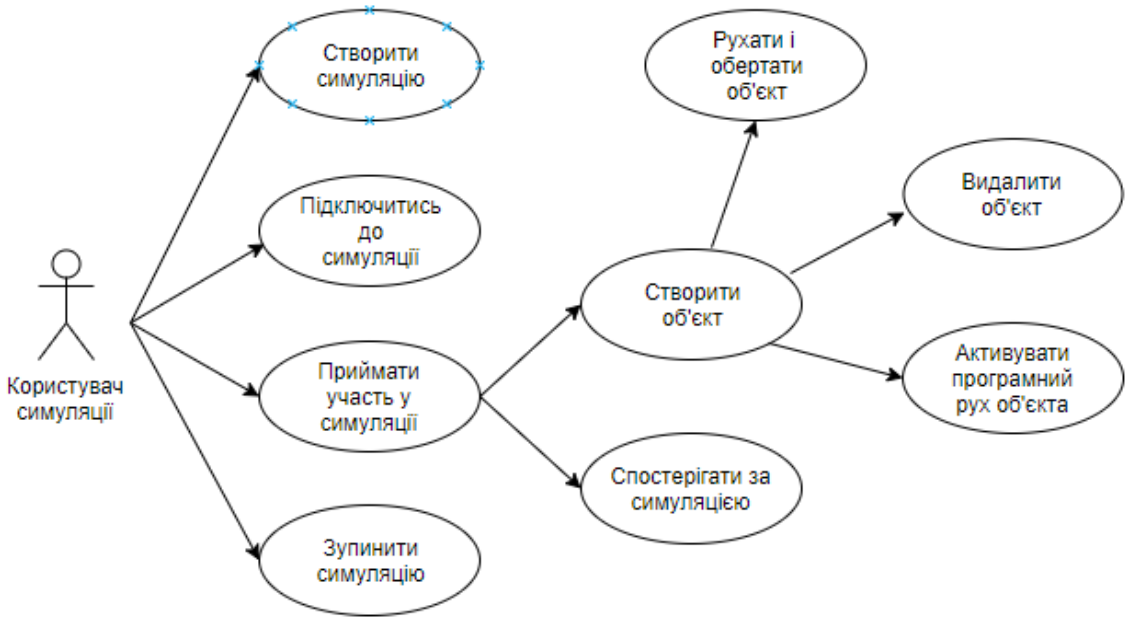


Рисунок 4.2 — Діаграма прецедентів системи

Користувач має можливість створити симуляцію і виступати при цьому мастер-клієнтом, підключитися до існуючої симуляції, приймати участь у симуляційному процесі та припинити поточну симуляцію. Під участю у симуляції мається на увазі або ж створення-видалення об’єкта та маніпуляції над ним, або ж спостерігання за симуляцією без створення і управління власного об’єкта.

4.3 Опис підсистеми повідомлень у застосунку

Було розроблено підсистему повідомлень для організації зручної комунікації між клієнтами системи (клієнт-клієнт, клієнт-сервер). Наприклад, при передачі повідомлення мережею існує метод в NetworkAPI void **sendUnicastMessage**(quint16 port, NetMessage& message, const QHostAddress& address). NetMessage – це базовий клас для усіх мережевих повідомлень. Виходячи з сигнатури методу, можна зробити висновок, що нас зовсім не цікавить, яке конкретно це повідомлення. Реалізовано за допомогою поліморфізму і наслідування – елементів об’єктно-орієнтованого програмування(ООП). Кожне повідомлення має свій тип, власний номер, може бути “надійним” або ні, тому ці характеристики зберігаються в класі NetMessage. Слід звернути увагу на віртуальні методи цього класу:

- virtual void ***serialize***(QDataStream& stream);
- virtual void ***deserialize***(QDataStream& stream);

Ці методи призначені для серіалізації та десеріалізації відповідно у і з stream. Іншими словами, конвертуємо наші конкретні повідомлення у бінарний вигляд та назад для передачі мережею. Класи-нащадки перевизначають ці методи з обов’язковим викликом базового методу в першу чергу.

Наведу приклад реалізації серіалізації в буфер:

```
QDataStream& operator <<(QDataStream& stream, NetMessage& message)
{
   message.serialize(stream);
   return stream;
}
```

Наперед невідомо, яке повідомлення необхідно буде серіалізувати, проте така реалізація цього і не потребує, і це дуже полегшує роботу.

Особливістю класу NetMessage є застосування у ньому породжувального патерну – Фабричний метод. За це відповідає наступна статична функція:
static std::unique_ptr<NetMessage> **createMessage**(MessageType type).
Як можна побачити, обов'язковим аргументом методу є тип повідомлення (усі можливі типи даної симуляції вказано у табл. 4.1), а значення, яке повертає метод – інтелектуальний показчик на об'єкт типу NetMessage. Це зручно, тому, що щойно об'єкт, що буде створений даним способом вийде за область видимості місця його створення, пам'ять буде автоматично вивільнена. Проте це необхідно мати на увазі.

Тип повідомлення MessageType – це строго-типізоване перерахування, яке не допускає неявної конвертації до цілого числа. Це змушує програміста явно вказувати область видимості MessageType для використання конкретного типу.

Усі повідомлення окрім StateSynchronizationMessage і RUDPAcknowledgement є "надійними", тобто ми гарантовано отримаємо їх.

Таблиця 4.1. Типи повідомлень у застосунку

Ім'я елемента перерахування	**Опис повідомлення**
NoneType	Невизначений тип
ConnectionRequestMessage	Клієнтський запит, спроба "підключитися" до сервера
ConnectionResponseMessage	Відповідь сервера, відправка інформації про існуючих клієнтів
UpdatePeerListMessage	Відповідь сервера, сповіщення клієнтів про появу нового клієнта
StateSynchronizationMessage	Синхронізація положення, орієнтації за номером об'єкта
ObjectCreationRequestMessage	Клієнтський запит на створення нового симуляційного об'єкта

Продовження таблиці 4.1

Ім’я елемента перерахування	**Опис повідомлення**
ObjectCreationResponseMessage	Відповідь сервера, дозвіл на створення об’єкта
ObjectCreationSynchronizationMessage	Відповідь сервера, сповіщення інших учасників симуляції про створення об’єкта
RUDPAcknowledgement	Підтвердження отримання «надійного» повідомлення

В підрозділі 3.3 було описано можливий потік взаємодії клієнтів і сервера. На рис 4.3 зображено один з можливих варіантів розвитку подій в контексті розроблених повідомлень, що описані у табл. 4.1.

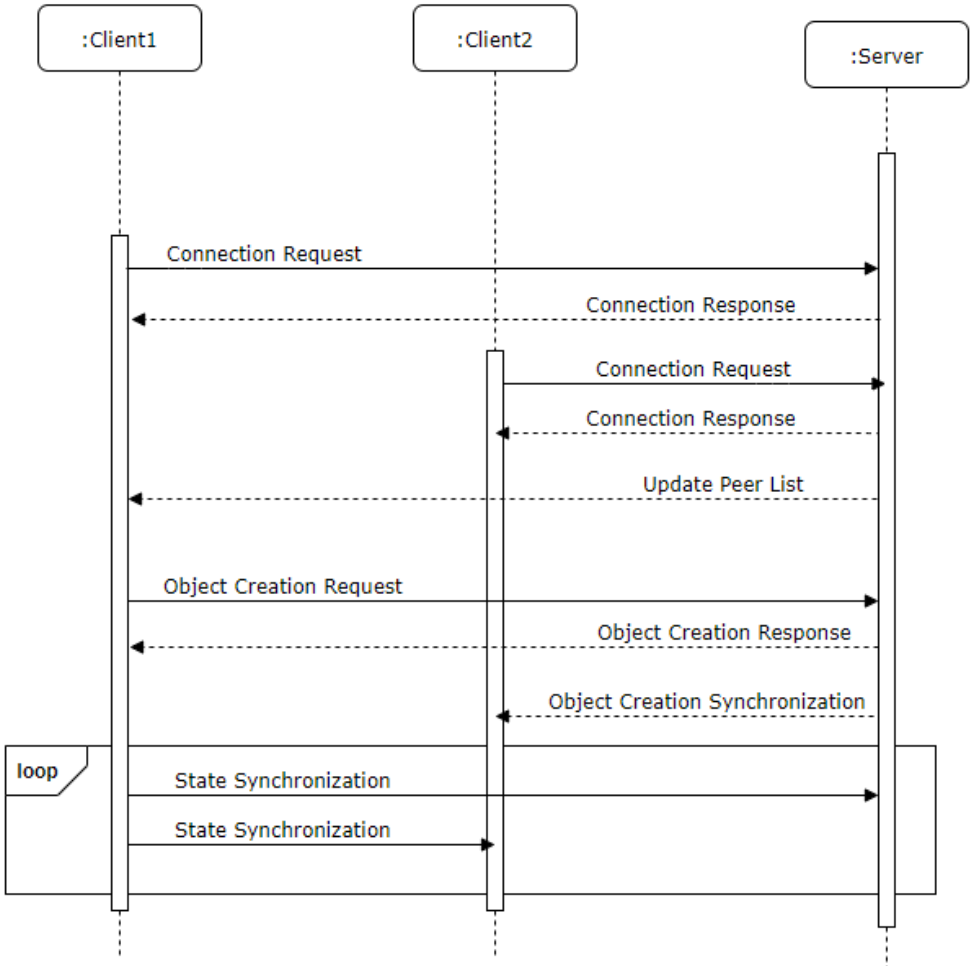


Рисунок 4.3 — Можливий варіант послідовності взаємодії клієнтів і сервера

Допускаємо, що сервер на цей момент вже працює. Перший клієнт підключається до симуляції, відправляється повідомлення серверу ConnectionRequestMessage. Сервер відповідає клієнту повідомленням ConnectionResponseMessage і додає учасника до списку клієнтів. Після цього моменту, клієнт може долучитися до симуляції, що детально було описано на рис 4.2, Далі підключається другий клієнт, також відправляє повідомлення ConnectionRequestMessage. Сервер відсилає клієнту відповідь, яка містить інформацію про першого учасника. Існуючому до цього моменту клієнту сервер відправляє повідомлення UpdatePeerListMessage з інформацією про нового учасника. Потім, перший учасник намагається створити фізичний об’єкт, відправляє повідомлення серверу ObjectCreationRequestMessage. Отримуючи це повідомлення сервер відправляє відповідь у вигляді ObjectCreationResponseMessage і створює фізичний об’єкт локально у себе. Другому учаснику сервер відправляє повідомлення ObjectCreationSynchronizationMessage, що б він також створив даний об’єкт, який буде синхронізуватися першим клієнтом. Повідомлення ObjectCreationResponseMessage слугує для першого клієнта підтвердженням або дозволом на створення об’єкта. Тепер клієнт кожний кадр симуляції відправлятиме стан свого об’єкта (його характеристики) у повідомленні ObjectCreationSynchronizationMessage усім іншим учасникам симуляції, в тому числі і серверу. В той момент коли будь-який з учасників отримує це повідомлення, він шукає у локальному списку об’єкт за унікальним номером та встановлює йому передану позицію і орієнтацію.

Повідомлення RUDPAcknowledgement використовується у реалізації RUDP і містить коротку інформацію, що однозначно ідентифікує повідомлення, доставку якого необхідно підтвердити.

4.4 Опис основного підходу гарантування доставки

Перш за все слід нагадати, що система є одноранговою, а тому кожен клієнт відправляє повідомлення іншим клієнтам. Описаний в розділі 2 метод гарантування доставки пакетів зумовлений організацією двох черг (для “надійних” і “ненадійних” повідомлень) і окремих порядкових номерів відповідно. Також слід зазначити, що відправка «надійного» повідомлення відбуватиметься до тих пір, доки ми не отримаємо підтвердження про його отримання, а тому їх необхідно деякий час зберігати. І останнє, якщо клієнт, якому було відправлено повідомлення з порушенням порядку, нам необхідно буферизувати і зберігати деякий час ці повідомлення перед обробкою програми. Наприклад, теоретично, клієнт може спробувати підключитися до системи, і відразу створити об’єкт (відправити запит на його створення). При цьому повідомлення про дозвіл створення об’єкта може дійти до клієнта раніше, ніж підтвердження підключення до сервера. В даній ситуації проблему порядку однозначно буде вирішено.

Розглядаючи усі ці моменти, я дійшов до наступного вигляду збереження цих даних у структурі PeerMessagesInfo відповідно для кожного клієнта:

```
struct PeerMessagesInfo
{
    // Receiving data
    unsigned int m_nextExpectedReliableMessageID = 0;
    unsigned int m_nextExpectedUnreliableMessageID = 0;
    QVector<quint64> m_reliableReceivingPacketNumbers;

    // Sending data
    unsigned int m_lastSentReliableMessageID = 0;
    unsigned int m_lastSentUnreliableMessageID = 0;
    QVector<quint64> m_reliableSendingPacketNumbers;
};
```

Тип QVector <quint64> описує динамічний масив цілих восьмибайтних чисел, номерів пакетів. А пакети зберігаються в хеш-таблиці, за власним порядковим номером. Коли приходить підтвердження про отримання від іншого клієнта, пакет видаляється з хеш-таблиці, і його номер з масиву пакетів також.

Реалізацію пакету на рівні RUDP буде подано у додатку Б:

4.5 Опис розробки юніт-тестів

Найсуттєвішою підсистемою розроблюваної технології, яка забезпечує «комунікацію» учасників симуляції є підсистема повідомлень. Вона використовується підсистемами Мережа (Network) і Симуляція (Simulation).

Усі повідомлення відрізняються своїми полями і даними, що зберігають у собі. Для відправки повідомлень мережею, їх необхідно серіалізувати, тобто конвертувати у бінарний вигляд. Після того, як буде сформовано масив байт на стороні отримувача, з потоку десеріалізується надіслане повідомлення. Необхідно перевірити коректність роботи серіалізації і десеріалізації повідомлень, збереження інформації, що несуть в собі повідомлення,

Було написано юніт-тести саме для цієї підсистеми тому, що правильність роботи збереження повідомлень, їх серіалізація та десеріалізація мають чітко виконувати свою роботу. Так, як усі повідомлення створюються через фабричний метод createMessage(MessageType), слід перевірити відповідність створеного повідомлення переданому типу. Ще одним пунктом, який неодмінно необхідно було перевірити, чи правильно система взагалі розподіляє повідомлення на “надійні” і “ненадійні”.

Успішний результат проходження юніт-тестування зображено на рис. 4.4.

```
PASS   : NetMessageFlowTest::initTestCase()
PASS   : NetMessageFlowTest::testMessageTypes()
PASS   : NetMessageFlowTest::testMessageReliability()
PASS   : NetMessageFlowTest::testSerializationStateSynchronizationMessage()
PASS   : NetMessageFlowTest::testSerializationObjectCreationResponseMessage()
PASS   : NetMessageFlowTest::cleanupTestCase()
Totals: 6 passed, 0 failed, 0 skipped, 0 blacklisted, 37ms
********* Finished testing of NetMessageFlowTest *********
```

Рисунок 4.4 — Результат юніт-тестування

4.6 Опис тестування продуктивності

Для перевірки мережевої частини розроблюваної системи піддано стресовому тестуванню, тобто перевірено при конкретних навантаженнях на систему. Було просимульовано мережеві проблеми за допомогою програми clumsy. Інтерфейс цієї програми зображено на рис. 4.5.

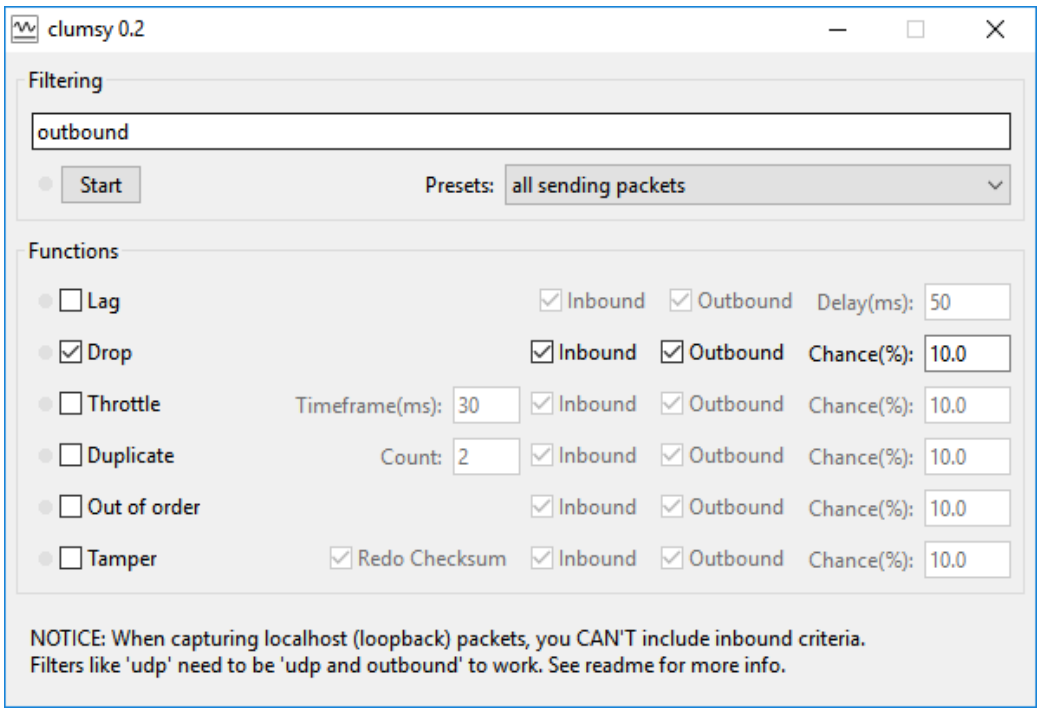


Рисунок 4.5 — Інтерфейс програми clumsy

Цей застосунок призначений для симуляції мережевої затримки, втрати та дублікації пакетів, втрата порядку пакетів, одним словом, проблем, які необхідно було вирішити для UDP при розробці RUDP.

5. МЕТОДИКА РОБОТИ КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Розроблена система синхронізації фізичних об’єктів має інтуїтивно простий і нескладний інтерфейс. Витримано мінімалістичний підхід у розробці графічного інтерфейсу. Програма розроблювалась у крос-платформному середовищі Qt Creator, а тому, окрім Windows, можливий її запуск на Unix-подібних системах і Mac.

5.1 Встановлення застосунку та системні вимоги

Для того, щоб почати користуватися розробленою системою, необхідно розпакувати архів, в якому буде знаходитися виконуваний exe-файл і відповідні динамічні бібліотеки, необхідні для запуску. Робота фізичної симуляції з використанням Box2D є досить оптимізованою, тому для підтримання достатньої продуктивності буде достатньо процесора на базі Intel Core i3. Необхідною вимогою є підключення до мережі Інтернет.

5.2 Документація до розроблюваної системи

Систему було документовано за допомогою програми Doxygen. Doxygen - це кросплатформна система, призначена для документування коду програм, яка підтримує більшість існуючих мов, зокрема C/C++, Objective-C, PHP, Python, Java, C#.

Doxygen генерує документацію на основі набору вихідних текстів і також може бути налаштований для вилучення структури програми з недокументованих вихідних текстів. Можливе складання графів залежностей програмних об'єктів, діаграм класів та вихідних кодів з гіперпосиланнями.

Код, написаний мовою C++ має наступний шаблон для задання документації:

```
/**
 * @brief Class NetworkAPI provides operations to work with socket and network initializations.
 */
class NetworkAPI : public QObject
```

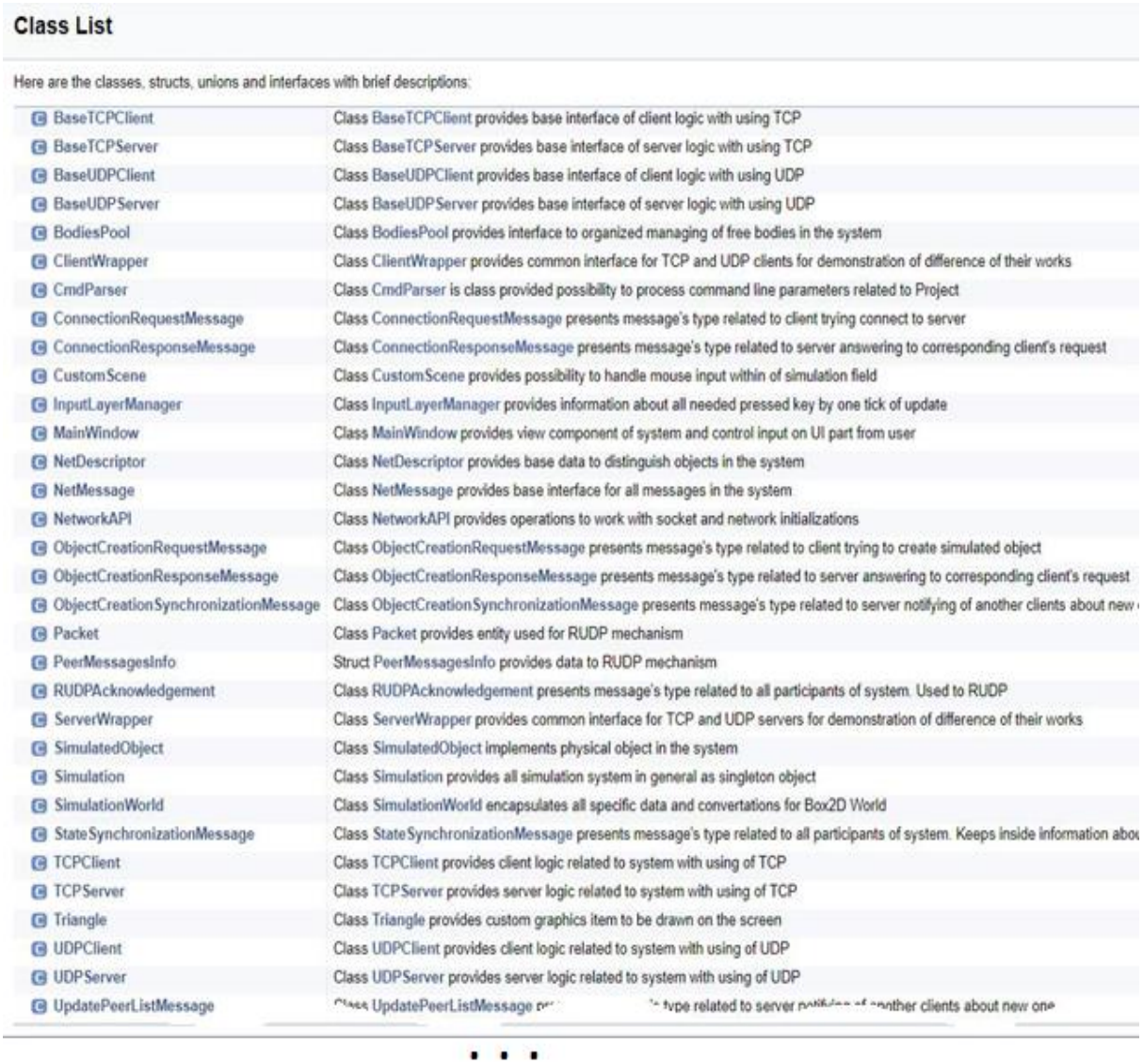
або

```
/**
    * @brief Initialize of network API layer
    * @param isAuthoritarianClient if \c true - init server logic; \c false - client
    * @param isRUDPEnabled Boolean value shown using RUDP in the system. if \c false - UDP will be used
    */
   void init(bool isAuthoritarianClient, bool isRUDPEnabled = true);
```

або

```
/**
    * @brief Returns \c true if --server-port cmd option is set; \c false otherwise.
    * @return bool
    */
   bool isServerPortSet() const;
```

Після коментування коду розроблюваної системи у стилі Doxygen, було за допомогою цього застосунку згенеровано файли документації у вигляді html-файлів. Відкривши файл index.html, після переходу до списку класів буде відображено список усіх структур і класів системи та коротке пояснення до них, як це можна побачити на рис 5.1. Це дає змогу відразу зрозуміти, для чого призначений той чи інший програмний клас. У лівій колонці відображається ім’я класів, у правій – текст, що розміщується за ключовим словом @brief у коді коментаря. Повний детальний опис буде розміщено, якщо розгорнути бажаний клас.



Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BaseTCPClient	Class BaseTCPClient provides base interface of client logic with using TCP
BaseTCPServer	Class BaseTCPServer provides base interface of server logic with using TCP
BaseUDPClient	Class BaseUDPClient provides base interface of client logic with using UDP
BaseUDPServer	Class BaseUDPServer provides base interface of server logic with using UDP
BodiesPool	Class BodiesPool provides interface to organized managing of free bodies in the system
ClientWrapper	Class ClientWrapper provides common interface for TCP and UDP clients for demonstration of difference of their works
CmdParser	Class CmdParser is class provided possibility to process command line parameters related to Project
ConnectionRequestMessage	Class ConnectionRequestMessage presents message's type related to client trying connect to server
ConnectionResponseMessage	Class ConnectionResponseMessage presents message's type related to server answering to corresponding client's request
CustomScene	Class CustomScene provides possibility to handle mouse input within of simulation field
InputLayerManager	Class InputLayerManager provides information about all needed pressed key by one tick of update
MainWindow	Class MainWindow provides view component of system and control input on UI part from user
NetDescriptor	Class NetDescriptor provides base data to distinguish objects in the system
NetMessage	Class NetMessage provides base interface for all messages in the system
NetworkAPI	Class NetworkAPI provides operations to work with socket and network initializations
ObjectCreationRequestMessage	Class ObjectCreationRequestMessage presents message's type related to client trying to create simulated object
ObjectCreationResponseMessage	Class ObjectCreationResponseMessage presents message's type related to server answering to corresponding client's request
ObjectCreationSynchronizationMessage	Class ObjectCreationSynchronizationMessage presents message's type related to server notifying of another clients about new
Packet	Class Packet provides entity used for RUDP mechanism
PeerMessagesInfo	Struct PeerMessagesInfo provides data to RUDP mechanism
RUDPAcknowledgement	Class RUDPAcknowledgement presents message's type related to all participants of system. Used to RUDP
ServerWrapper	Class ServerWrapper provides common interface for TCP and UDP servers for demonstration of difference of their works
SimulatedObject	Class SimulatedObject implements physical object in the system
Simulation	Class Simulation provides all simulation system in general as singleton object
SimulationWorld	Class SimulationWorld encapsulates all specific data and convertations for Box2D World
StateSynchronizationMessage	Class StateSynchronizationMessage presents message's type related to all participants of system. Keeps inside information abo
TCPClient	Class TCPClient provides client logic related to system with using of TCP
TCPServer	Class TCPServer provides server logic related to system with using of TCP
Triangle	Class Triangle provides custom graphics item to be drawn on the screen
UDPClient	Class UDPClient provides client logic related to system with using of UDP
UDPServer	Class UDPServer provides server logic related to system with using of UDP
UpdatePeerListMessage	Class UpdatePeerListMessage ... type related to server ... another clients about new one

Рисунок 5.1 — Частина списку класів у документації системи

Документацію у повному обсязі було оформлено англійською мовою. На рис. 5.2 зображено опис конкретного класу NetworkAPI: коротка інформація про клас, заголовочний файл, який необхідно підключити для використання цього класу, невелика діаграма, що відображає ієрархію наслідування для поточного класу (NetworkAPI наслідується від QObject), публічні слоти, сигнали, якщо вони присутні, публічні і приватні члени класу. Кожен метод або поле детально описані, для того, щоб у подальшому не виникало жодних проблем для використання функціоналу будь-якого класу.

Public Slots | Signals | Public Member Fu

NetworkAPI Class Reference

Class **NetworkAPI** provides operations to work with socket and network initializations. More...

`#include <networkapi.h>`

Inheritance diagram for NetworkAPI:

Public Slots

void **readReady** ()
Slot triggered when readyRead() signal emitted.

Signals

void **signalMessageForUI** (const QString message)
Signal is emiited for visual debugging. More...

Public Member Functions

void **init** (bool **isAuthoritarianClient**, bool isRUDPEnabled=true)
Initialize of network API layer. More...

void **shutdown** ()
Clear all data for single instance explicitly.

bool **isAuthoritarianClient** () const
Returns `true` if server logic was inititialized; `false` - client one. More...

void **setIsAuthoritarianClient** (bool **isAuthoritarianClient**)
Sets server flag for network API. More...

std::unique_ptr< **UDPServer** > & **getServer** ()
Returns Server object which initialized when isAuthoritarianClient - `true`. More...

std::unique_ptr< **UDPClient** > & **getClient** ()
Returns Client object which initialized when isAuthoritarianClient - `false`. More...

void **startListen** ()
Binds socket for opening(Server specific method).

• • •

Рисунок 5.2 — Частина списку класів у документації системи

5.3 Інструкція з використання програмного продукту

Запуск програмного застосунку зображено на рис. 5.3. У верхніх вкладках меню можна вибрати пункт “Ініціалізувати” або “Допомога”. У лівій частині вікна знаходиться власне поле для симуляції фізичних об’єктів, у правій –

відображатимуться повідомлення, що відсилаються або приймаються даним учасником симуляції. Це зроблено, як для відладки та тестування розроблюваної системи, так і для демонстрації результатів роботи. На рис. 5.4 вказано елементи пункту меню “Ініціалізувати”. Серед них “Створити симуляцію” (серверна поведінка), “Долучитися до симуляції” (клієнтна поведінка), “Очистити симуляцію” (повернутися до початкового стану програми) і “Вихід”.

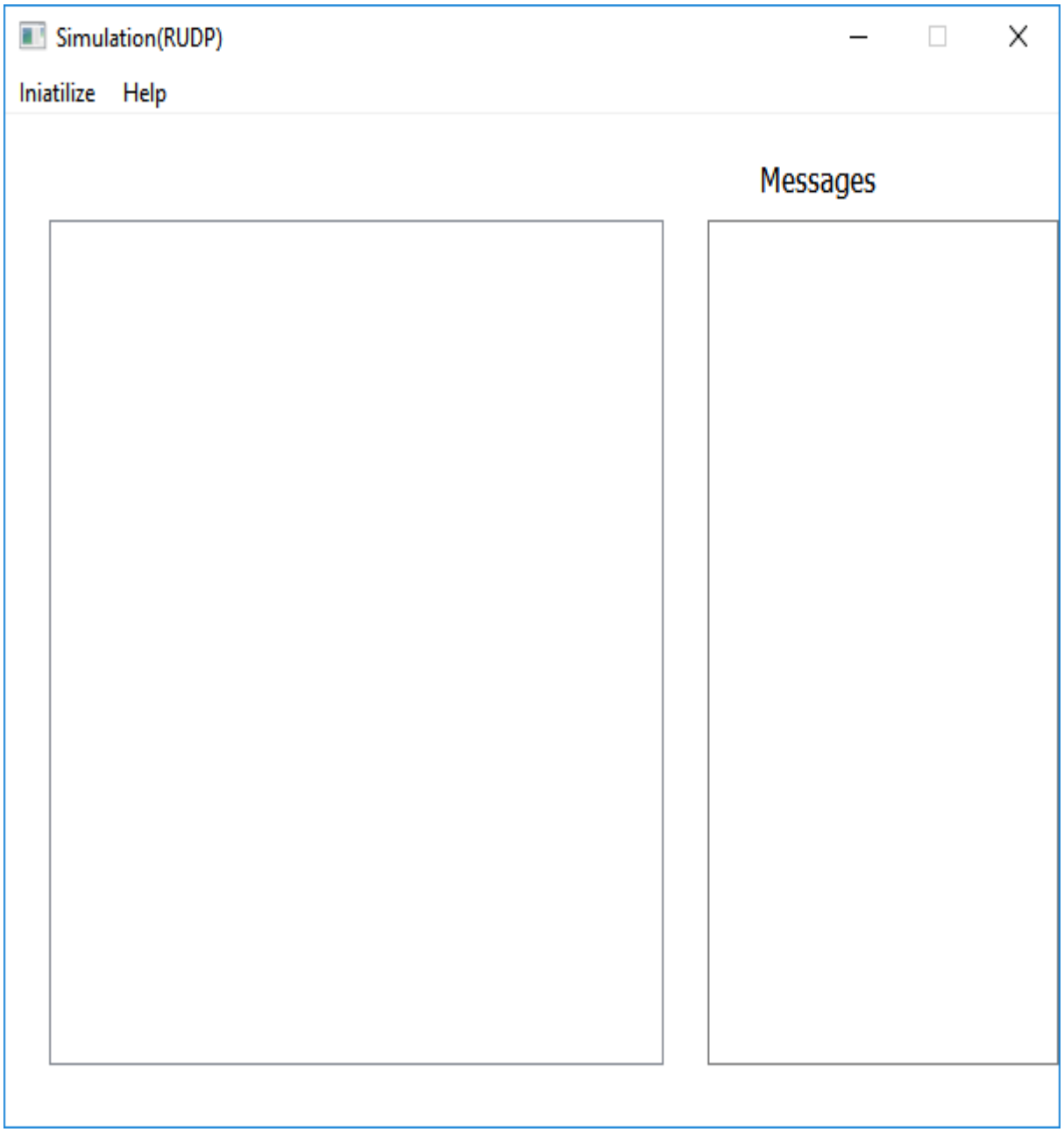


Рисунок 5.3 — Головне вікно системи

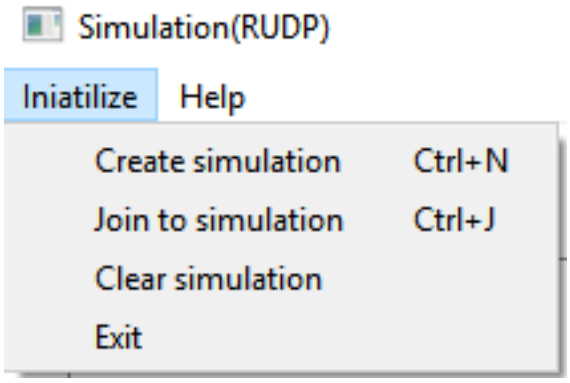


Рисунок 5.4 — Елементи пункту меню “Ініціалізувати”

В пункті “Допомога” єдиний елемент, “Як користуватися” (рис. 5.5).

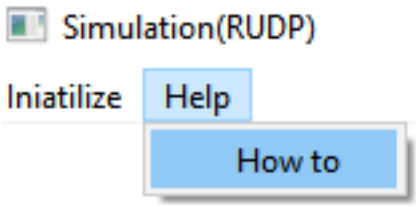


Рисунок 5.5 — Елемент пункту меню “Допомога”

Було додано гарячі клавіші: для створення симуляції – Ctrl + N, для долучення до симуляції Ctrl + J.

Після того, як користувач розпочав або підключився до симуляції, він може створити власний об’єкт, натиснувши праву кнопку миші і рухати його за допомогою клавіш W, A, S, D. Результат таких дій зображено на рис. 5.6.

На рис. 5.7 зображено приклад синхронізації двох клієнтів, кожен з яких створив власний об’єкт і рухає його по симуляційному полю. Симуляція виконується на локальному хості.

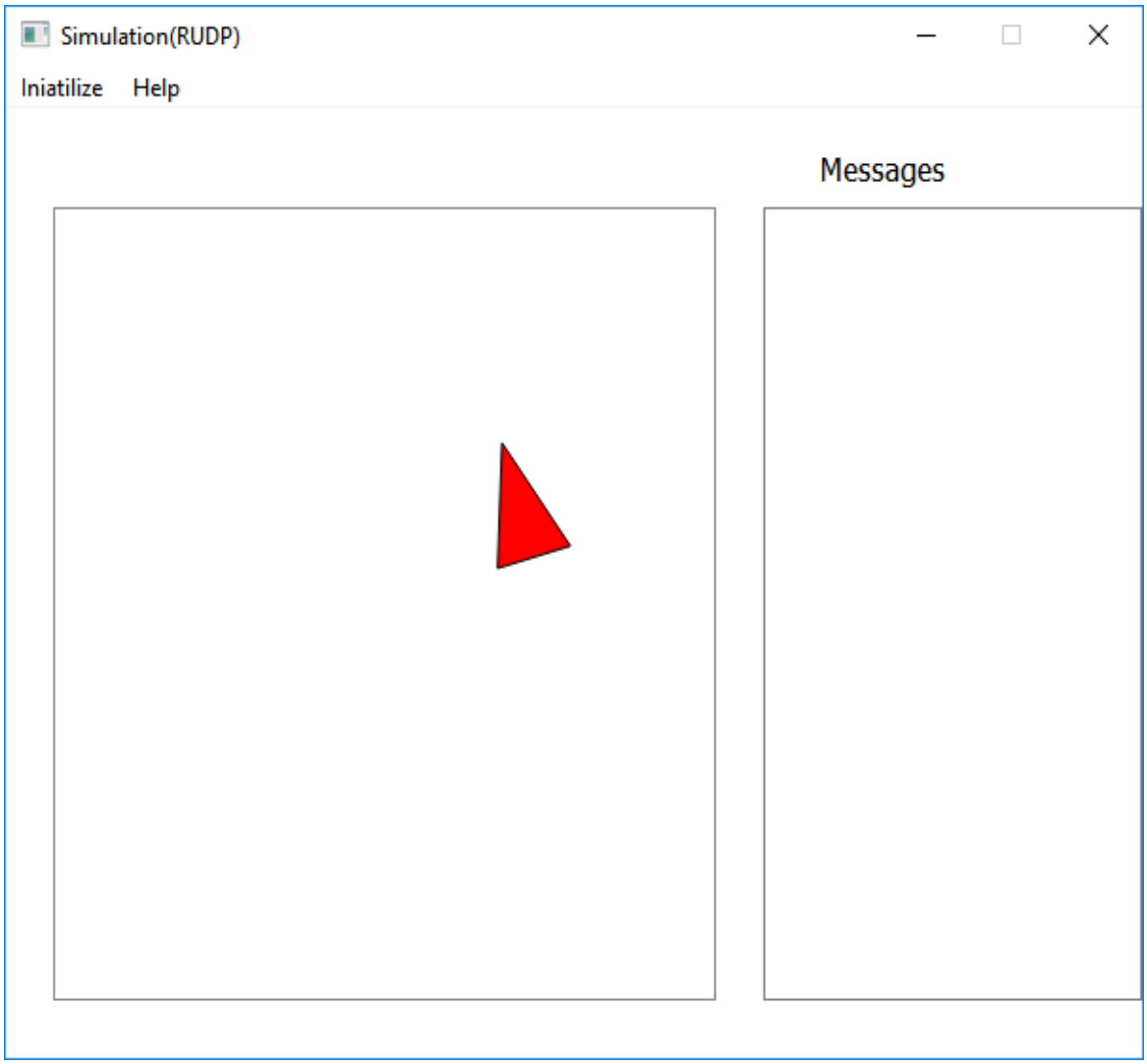


Рисунок 5.6 — Приклад зміни фізичних характеристик створеного об’єкта

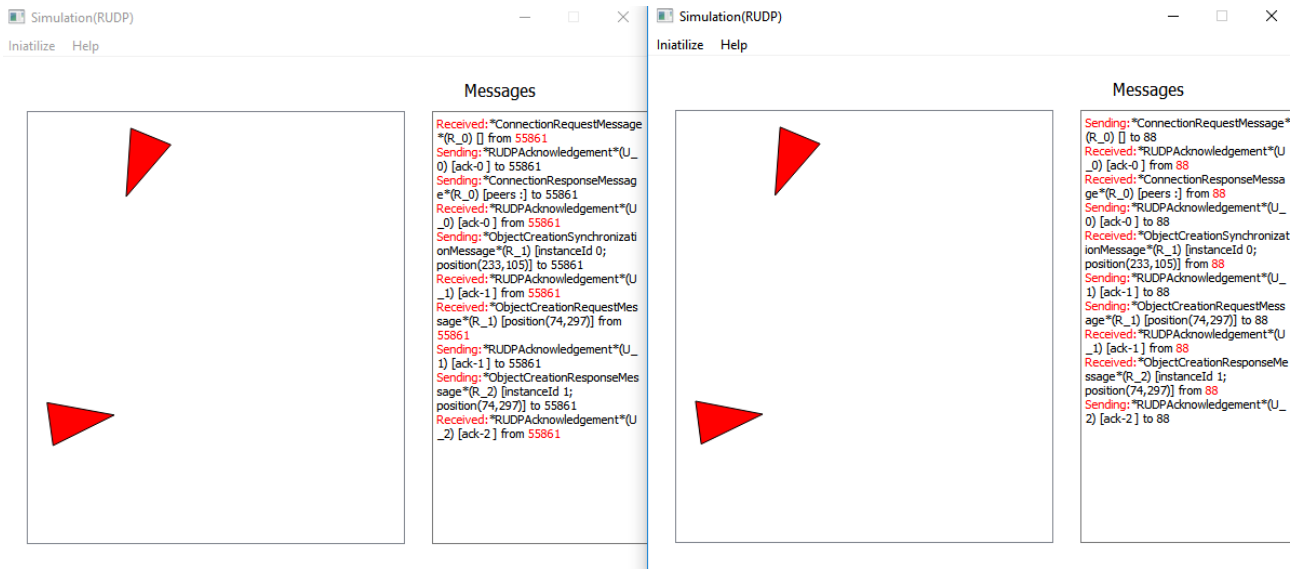


Рисунок 5.7 — Приклад синхронізації двох клієнтів на localhost

На рис. 5.8 відтворено “підключення” клієнту до серверу і створення об’єктів, проте за умови симуляції втрати пакетів у п’ятдесят відсотків за допомогою програмного додатку clumsy-0.2-win64, який симулює також затримку доставлення пакетів, їх дублікацію, порушення порядку відправлення.

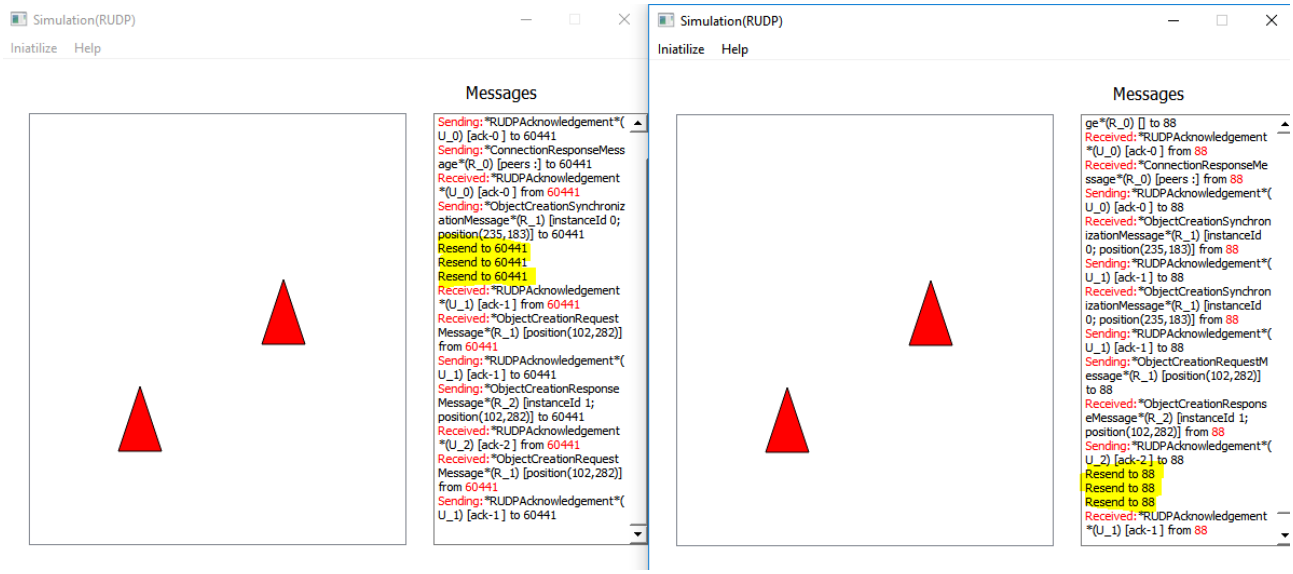


Рисунок 5.8 — Приклад синхронізації двох клієнтів на localhost з втратою пакетів у п’ятдесят відсотків

Можна побачити, що учасники симуляції дійсно за відсутності підтвердження намагаються відправити пакет повторно. Кількість спроб вказано в конфігураційному файлі системи і дорівнює восьми.

ВИСНОВКИ

У ході проходження практики було розроблено фізичну симуляцію, яка стала основою системи для тестування і порівняння роботи протоколів TCP, UDP і RUDP, збільшуючи вірогідність проблем з мережею.

Було доведено, що симуляція з використанням TCP втрачатиме плавність навіть при двох відсотках втрат пакетів. При використанні протоколу UDP відповідальність за обробку помилок і повторну передачу даних покладена на протоколи вищого рівня. Протокол UDP є ефективним для серверів, що надсилають невеликі відповіді великій кількості клієнтів. Але його недоліком для даної системи є значний час, що потрібен на опрацювання помилок при передачі даних і, взагалі, втрата необхідних для системи пакетів.

У ході програмування зазначеної клієнт-серверної системи використано протокол, який поєднує в собі ефективність UDP і надійність TCP - RUDP. Його використовують для забезпечення надійної передачі даних між пакетно-орієнтованими застосунками. На відміну від UDP, RUDP забезпечує такі функції, як підтвердження доставки пакетів, повторну відправку втрачених пакетів. Важливо зазначити, що клієнт застосунку може самостійно вирішувати у яких випадках, йому необхідна гарантія доставки його повідомлень.

Програмний застосунок було тестовано за допомогою програми, що симулює втрати пакетів під час відправки, їх дублікацію і порушення порядку. Не дивлячись на усі ці пункти, система чудово впоралася з цим.

Отже, проведена практика закріпила навички проектування систем та побудови архітектури, дозволила застосувати набуті вміння професійної діяльності, покращила розуміння клієнт-серверних систем, конкретних протоколів, особливо в порівнянні один з одним в реальній системі. Було проаналізовано декілька алгоритмів забезпечення надійної доставки пакетів, і реалізовано найоптимальніший з них.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C. Partridge, R. Hinden: Specification RFC 1151 Experimental - Version 2 of the Reliable Data Protocol (RDP) (1990)

2. State Synchronization. Keeping simulations in sync. [Електроннний ресурс] – Режим доступу: https://gafferongames.com/post/state_synchronization/

3. Reliable UDP Algorithms. [Електроннний ресурс] – Режим доступу: https://io7m.com/documents/udp-reliable/

4. Страуструп Б. Язык программирования C++. Специальное издание = The C++ programming language. Special edition. — М.: Бином-Пресс, 2007. — 1104с. —ISBN 5-7989-0223-4

5. Архитектурные шаблоны и стили. [Електроннний ресурс] – Режим доступу: https://studfiles.net/preview/6413815/page:11/

6. Qt documentation. Signals & Slots. [Електроннний ресурс] – Режим доступу: https://doc.qt.io/qt-5/signalsandslots.html

7. Qt documentation. QUDPSocket Class. [Електроннний ресурс] – Режим доступу: https://doc.qt.io/qt-5/qudpsocket.html

ДОДАТОК А

Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191__18Б

Аркушів 1

Київ 2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_18Б 83-1	Записка.docx	Пояснювальна записка
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_18Б 14-1	Копія публікації.docx	Копія публікації у студентській конференції
Компоненти		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_18Б 12-1	SynchronizingPhysicsSimulations.pro	Програмний модуль

ДОДАТОК Б

Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++

Текст програмного модуля

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_ 18Б 12-2

Аркушів 12

Київ 2019

```
// main.cpp
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // Process the actual command line arguments given by the user
    CmdParser::getInstance().parseAppArgs(a);

    MainWindow w;
    w.show();

    return a.exec();
}


// mainwindow.h
/**
 * @brief Class MainWindow provides view component of system and control input on UI part from user.
 */
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    /**
     * @brief Constructs MainWindow instance
     * @param parent Pointer to QWidget
     */
    explicit MainWindow(QWidget *parent = nullptr);
    /**
     * @brief Destructs MainWindow instance
     */
    ~MainWindow();

public slots:
    /**
     * @brief Slot triggered when sendMessageForUI() signal emitted
     * @param message String value need to show
     */
    void handleMessageForUI(const QString message);
    /**
     * @brief Slot triggered when sendObserverList() signal emitted
     * @param list List of client's names
     */
    void handleObserverListForUI(const QStringList& list);
    /**
     * @brief Slot triggered when sendPhysicalQuantitiesForView() signal emitted
     *
     * @param instanceID Object's unique number
     * @param x Object's x of position on simulation field
     * @param y Object's y of position on simulation field
     * @param rotation Object's rotation in degree
     */
    void handlePhysicalQuantities(quint16 instanceID, qreal x, qreal y, qreal rotation);

private slots:
    /**
     * @brief Slot triggered when user press Create simulation button
     */
    void on_actionCreate_simulation_triggered();
    /**
     * @brief Slot triggered when user press Join to simulation button
     */
    void on_actionJoin_to_simulation_triggered();
    /**
     * @brief Slot triggered when user press Exit button
     */
    void on_actionExit_triggered();

    /**
     * @brief Slot triggered when signalNewObjectCoordinate() signal emitted
     * @param point QPointF value presented object's position
     */
    void requestMasterObjectInPoint(QPointF point);
    /**
     * @brief Slot triggered when signalConfirmedCreationMasterObject() signal emitted
```

```
    * @param instanceID New object's unique number
    * @param position New object's position
    */
   void handleResponseMasterObjectCreation(quint16 instanceID, QPointF position);
   /**
    * @brief Slot triggered when signalCreationReplicaObject() signal emitted
    * @param instanceID New object's unique number
    * @param position New object's position
    */
   void handleResponseReplicaObjectCreation(quint16 instanceID, QPointF position);
   /**
    * @brief Slot triggered when signalObjectCreationSynchronization() signal emitted
    * @param requestorPort Port of peer requested creation of object
    * @param position Requested object's position
    */
   void handleOobjectCreationSynchronization(quint16 requestorPort, QPointF position);
   /**
    * @brief  Slot triggered when synchronizeSimulatedState() signal emitted
    * @param instanceID Object's instance id
    * @param x Object's x position
    * @param y Object's y position
    * @param rotation Object's rotation
    */
   void synchronizeSimulatedObject(quint16 instanceID, qreal x, qreal y, qreal rotation);

   /**
    * @brief Slot triggered when user press Clear simulation button
    */
   void on_actionClear_simulation_triggered();
   …
};


// netmessage.h
/**
 * @brief Class NetMessage provides base interface for all messages in the system.
 *
 * Created for wrapping custom data in structured view and creation of single interface
 * to send our data by network. createMessage(type) method implemented as Factory Method.
 * Class has methods serialize(stream) and deserialize(stream) for convert message to
 * binary format. Class NetMessage also has virtual method toString(). All enumerated methods
 * may be overriden in inheritors.
 * All messages have own messageID, can be reliable or unreliable in context of this system.
 */
class NetMessage
{
public:
   /**
    * @brief Constructs base part of message.
    * @param type Message's type
    */
   NetMessage(MessageType type = MessageType::NoneType);
   /**
    * @brief Destructs itself and inheritor's part.
    *
    * Marked as virtual to call inheritor's destructor.
    */
   virtual ~NetMessage();

   /**
    * @brief Returns message by concrete type.
    *
    * Implemented as Factory Method pattern. Returns unique_ptr.
    * Be careful, when this smart pointer go out of visibility's area,
    * object will destoy. You can take the raw pointer from it.
    *
    * @param type Message's type we need to create
    * @return std::unique_ptr<NetMessage>
    */
   static std::unique_ptr<NetMessage> createMessage(MessageType type);

   /**
    * @brief Serialize message to \a stream.
    *
    * Marked as virtual and may be overriden in inheritors.
    *
    * @param stream Reference to QDataStream
    */
   virtual void serialize(QDataStream& stream);
```

```
    /**
     * @brief Deserialize message from \a stream.
     *
     * Marked as virtual and may be overriden in inheritors.
     *
     * @param stream Reference to QDataStream
     */
    virtual void deserialize(QDataStream& stream);

    // for debug and logging
    /**
     * @brief Returns string view of current message.
     *
     * Marked as virtual and may be overriden in inheritors.
     * Used for logging and debug.
     *
     * @return QString
     */
    virtual QString toString() const;

    /**
     * @brief Returns message's type
     * @return MessageType
     */
    MessageType getType() const { return m_type; }

    // RUDP
    /**
     * @brief Returns realibility message's flag.
     * Returns \c true if message marked as reliable; \c false otherwise.
     *
     * @return bool
     */
    bool isReliable() const;
    /**
     * @brief Sets \a isReliable message's flag.
     *
     * @param isReliable Boolean value
     */
    void setIsReliable(bool isReliable);

    /**
     * @brief Returns message's id
     * @return unsigned int
     */
    unsigned int getMessageID() const;
    /**
     * @brief Sets message's id
     * @param messageID
     */
    void setMessageID(unsigned int messageID);

private:
    MessageType m_type; /**< Message's type */

    // RUDP realization
    bool m_isReliable; /**< Message's reliability flag */
    unsigned int m_messageID; /**< Message's id */
};

/**
 * @brief Overriden operator << to simple serialize of message to stream.
 *
 * Realized as open function. Use next way: stream << message;
 *
 * @param stream Reference to QDataStream
 * @param message Reference to NetMessage
 * @return QDataStream &operator
 */
QDataStream& operator <<(QDataStream& stream, NetMessage& message);
/**
 * @brief Overriden operator >> to simple deserialize of message from stream.
 * @param stream Reference to QDataStream
 * @param message Reference to NetMessage
 * @return QDataStream &operator >>
 */
QDataStream& operator >>(QDataStream& stream, NetMessage& message);
```

```
// netmessage.cpp
NetMessage::NetMessage(MessageType type)
   : m_type(type)
   , m_isReliable(false)
   , m_messageID(0)
{
}

std::unique_ptr<NetMessage> NetMessage::createMessage(MessageType type)
{
   switch(type)
   {
      case MessageType::ConnectionRequestMessage:
         return std::make_unique<ConnectionRequestMessage>();
      case MessageType::ConnectionResponseMessage:
         return std::make_unique<ConnectionResponseMessage>();
      case MessageType::StateSynchronizationMessage:
         return std::make_unique<StateSynchronizationMessage>();
      case MessageType::ObjectCreationRequestMessage:
         return std::make_unique<ObjectCreationRequestMessage>();
      case MessageType::ObjectCreationResponseMessage:
         return std::make_unique<ObjectCreationResponseMessage>();
      case MessageType::ObjectCreationSynchronizationMessage:
         return std::make_unique<ObjectCreationSynchronizationMessage>();
      case MessageType::UpdatePeerListMessage:
         return std::make_unique<UpdatePeerListMessage>();
      case MessageType::RUDPAcknowledgement:
         return std::make_unique<RUDPAcknowledgement>();
   }
   return nullptr;
}

void NetMessage::serialize(QDataStream& stream)
{
   stream << static_cast<qint8>(m_type) <<
           m_isReliable <<
           m_messageID;
}

void NetMessage::deserialize(QDataStream& stream)
{
   // Without type, because it processed manually for factory method using
   stream >> m_isReliable >>
           m_messageID;
}

QString NetMessage::toString() const
{
   const QString reliableLabel = isReliable() ? "R" : "U";
   return "(" + reliableLabel + "_" + QString::number(getMessageID()) + ") ";
}
bool NetMessage::isReliable() const
{
   return m_isReliable;
}
void NetMessage::setIsReliable(bool isReliable)
{
   m_isReliable = isReliable;
}
unsigned int NetMessage::getMessageID() const
{
   return m_messageID;
}
void NetMessage::setMessageID(unsigned int messageID)
{
   m_messageID = messageID;
}
QDataStream& operator <<(QDataStream& stream, NetMessage& message)
{
   message.serialize(stream);
   return stream;
}

QDataStream& operator >>(QDataStream& stream, NetMessage& message)
{
   message.deserialize(stream);
   return stream;
}
```

```
// simulation.cpp
bool Simulation::s_isSimulationStarted = false;

Simulation::Simulation(QObject *parent)
    : QObject(parent)
{
    m_simulationTimer = new QTimer();

    connect(m_simulationTimer, &QTimer::timeout, this, &Simulation::runFrame);

    m_simulationTimer->setInterval(FrameTimeMSec);

    m_objects.reserve(20);

    m_isUpdatable = true;
}
bool Simulation::getIsUpdatable() const
{
    return m_isUpdatable;
}
void Simulation::setIsUpdatable(bool isUpdatable)
{
    m_isUpdatable = isUpdatable;
}
bool Simulation::isStarted()
{
    return s_isSimulationStarted;
}
void Simulation::setAsStarted(const bool isStarted)
{
    s_isSimulationStarted = isStarted;
}

void Simulation::runFrame()
{
    // MAIN UPDATE
    if (!m_isUpdatable)
    {
        return;
    }

    preUpdate();
    update();
    render();
    postUpdate();
}

Simulation &Simulation::getInstance()
{
    static Simulation instance;
    return instance;
}

void Simulation::init()
{
    InputLayerManager::getInstance().init();
    SimulationWorld::getInstance().initBox2DSimulation();

    s_isSimulationStarted = true;
}

void Simulation::shutdown()
{
    m_simulationTimer->stop();
    m_objects.clear();

    InputLayerManager::getInstance().destroy();
    SimulationWorld::getInstance().destroyBox2DSimulation();

    s_isSimulationStarted = false;
}

void Simulation::startSimulation()
{
    m_simulationTimer->start();
}

SimulatedObject *Simulation::createSimulatedObject(bool isMaster)
```

```
{
    static quint16 objectInstanceID = 0;
    auto simulatedObject = std::make_unique<SimulatedObject>(isMaster, objectInstanceID);
    auto& bodiesPool = SimulationWorld::getInstance().getBodiesPool();
    assert(bodiesPool);

    b2Body* body = bodiesPool->getFirstFree();
    assert(body && "Pool is empty");

    simulatedObject->setSimulationBody(body);
    body->SetActive(true);
    m_objects.push_back(std::move(simulatedObject));

    ++objectInstanceID;

    return m_objects.back().get();
}

SimulatedObject *Simulation::getObjectByInstanceID(quint16 instanceID) const
{
    auto findIt = std::find_if(m_objects.begin(), m_objects.end(),
                               [&instanceID] (const auto& object)
    {
      return object->getInstanceID() == instanceID;
    });

    if (findIt != m_objects.end())
    {
       return findIt->get();
    }

    return nullptr;
}

void Simulation::preUpdate()
{
    InputLayerManager::getInstance().processInput();
}

void Simulation::update()
{
    if (QApplication::focusWidget())
    {
       for (auto& object : m_objects)
       {
          object->update();
       }
    }

    auto world = SimulationWorld::getInstance().getWorld();
    assert(world && "Null world");

    // Physics calculation
    world->Step(FrameTimeSec, 8, 3);
    world->ClearForces();

    // Synchronize master object's state
    for (auto& object : m_objects)
    {
       if (object->isMaster())
       {
          const quint16 instanceID = object->getInstanceID();
          QPointF realPos = object->getRealPosition();
          qreal angleDeg = object->getObjectRealAngle();

          emit synchronizeSimulatedState(instanceID, realPos.x(), realPos.y(), angleDeg);

          // Only ONE master object
          return;
       }
    }
}

void Simulation::render()
{
    for (auto& object : m_objects)
    {
       object->render();
    }
```

```
}

void Simulation::postUpdate()
{
    InputLayerManager::getInstance().resetCurState();
}


// networkapi.cpp
quint16 NetworkAPI::ServerPort = 88;

NetworkAPI& NetworkAPI::getInstance()
{
    static NetworkAPI instance;
    return instance;
}

void NetworkAPI::init(bool isAuthoritarianClient, bool isRUDPEnabled)
{
    m_isAuthoritarianClient = isAuthoritarianClient;
    m_isRUDPEnabled = isRUDPEnabled;

    if (m_isAuthoritarianClient)
    {
        m_server = std::make_unique<UDPServer>();
    }
    else
    {
        m_client = std::make_unique<UDPClient>();
    }
}

void NetworkAPI::shutdown()
{
    closeSocket();

    m_isAuthoritarianClient = false;
    if (m_server)
    {
        m_server.reset();
    }
    if (m_client)
    {
        m_client.reset();
    }

    m_messagesInfo.clear();

    for (Packet* packet : m_allPackets)
    {
        delete packet;
    }
    m_allPackets.clear();
}
```

```
NetworkAPI::NetworkAPI(QObject* parent)
   : QObject (parent)
   , m_isAuthoritarianClient(false)
   , m_isRUDPEnabled(false)
{
}

void NetworkAPI::WriteData(QByteArray& data, const QHostAddress& address, quint16 port)
{
   m_socket.writeDatagram(data, address, port);
}

void NetworkAPI::processMessage(NetMessage& message, const QHostAddress& addr, quint16 port)
{
   if (m_isAuthoritarianClient)
   {
      m_server->onProcessMessage(message, addr, port);
   }
   else
   {
      m_client->onProcessMessage(message, addr, port);
   }
}

void NetworkAPI::observeReceivedMessage(NetMessage& message, const QHostAddress& addr, quint16 port)
{
   // Receiving messages
   if (!m_messagesInfo.contains(port))
   {
      m_messagesInfo[port] = new PeerMessagesInfo();
   }
   PeerMessagesInfo* peerInfo = m_messagesInfo[port];

   if (message.getType() == MessageType::RUDPAcknowledgement)
   {
      auto findIt = std::find_if(m_allPackets.keyValueBegin(),
                                 m_allPackets.keyValueEnd(),
                                 [&message, &port] (const auto& keyValue)
      {
        return keyValue.second->m_port == port &&
               keyValue.second->m_messageID == message.getMessageID();
      });

      // README Here can be situation when we send ack to client,
      // and client has not receive this message yet.
      if (findIt != m_allPackets.keyValueEnd())
      {
         const quint64 packetID = (*findIt).first;

         // find message which we need to remove from buffer
         peerInfo->m_reliableSendingPacketNumbers.removeOne(packetID);

         // stop timer and delete packet from allPackets
```

```
            m_allPackets[packetID]->confirmDelivery();
            delete m_allPackets[packetID];
            m_allPackets.erase(findIt.base());
        }
    }
    else // actual message
    {
        if (message.isReliable())
        {
            // Need to send acknowledgement IN ANY WAY
            RUDPAcknowledgement ack;
            ack.setMessageID(message.getMessageID());
            sendUnicastMessage(port, ack, addr);

            /// Compare expected message id and new message
            // Reordering
            if (peerInfo->m_nextExpectedReliableMessageID < message.getMessageID())
            {
                // Need to buffer and NO process
                // Create packet
                Packet* packet = new Packet();
                packet->m_packetID = m_newPacketID++;
                packet->initMessageData(message);
                packet->m_address = addr;
                packet->m_port = port;
                packet->m_messageID = message.getMessageID();

                m_allPackets.insert(packet->m_packetID, packet);

                peerInfo->m_reliableReceivingPacketNumbers.push_back(packet->m_packetID);
            }
            else if (peerInfo->m_nextExpectedReliableMessageID == message.getMessageID())
            {
                if (peerInfo->m_reliableReceivingPacketNumbers.empty())
                {
                    ++peerInfo->m_nextExpectedReliableMessageID;
                    // Process one message
                    processMessage(message, addr, port);
                }
                else
                {
                    //need to push new packet value
                    // Create packet
                    Packet* packet = new Packet();
                    packet->m_packetID = m_newPacketID++;
                    packet->initMessageData(message);
                    packet->m_address = addr;
                    packet->m_port = port;
                    packet->m_messageID = message.getMessageID();

                    m_allPackets.insert(packet->m_packetID, packet);

                    peerInfo->m_reliableReceivingPacketNumbers.push_back(packet->m_packetID);
```

```
            std::sort(peerInfo->m_reliableReceivingPacketNumbers.begin(),
                    peerInfo->m_reliableReceivingPacketNumbers.end(),
                    [] (const auto& a, const auto& b)
            {
                return a < b;
            });

            QVector<quint64> packetsToDelete;
            // Process messages in right order
            for (quint64 packetID : peerInfo->m_reliableReceivingPacketNumbers)
            {
                assert(m_allPackets.contains(packetID) && "No such packet");
                Packet* curPacket = m_allPackets[packetID];
                if (peerInfo->m_nextExpectedReliableMessageID == curPacket->m_messageID)
                {
                    ++peerInfo->m_nextExpectedReliableMessageID;
                    std::unique_ptr<NetMessage> localMessage =
                            SerializationMessageHelper::deserializeMessageFromByteArray(curPacket->m_messageData);
                    processMessage(*localMessage, addr, port);

                    packetsToDelete.push_back(curPacket->m_packetID);
                }
                else
                {
                    break;
                }
            }

            // Delete processed messages from buffer
            for (quint64& packetID : packetsToDelete)
            {
                if (m_allPackets.contains(packetID))
                {
                    delete m_allPackets[packetID];
                    m_allPackets.remove(packetID);
                }
                peerInfo->m_reliableReceivingPacketNumbers.removeOne(packetID);
            }
        }
    }
    else if (peerInfo->m_nextExpectedReliableMessageID > message.getMessageID())
    {
        // Discard this message
        return;
    }
}
else // message is unreliable
{
    // Need to cmp expected value
    if (peerInfo->m_nextExpectedUnreliableMessageID <= message.getMessageID())
    {
        peerInfo->m_nextExpectedUnreliableMessageID = message.getMessageID() + 1;
        // Process by simulation
```

```
                processMessage(message, addr, port);
            }
        }
    }
}

void NetworkAPI::observeSendingMessage(NetMessage& message, const QHostAddress& addr, quint16 port)
{
    if (!m_messagesInfo.contains(port))
    {
        m_messagesInfo[port] = new PeerMessagesInfo();
    }
    PeerMessagesInfo* peerInfo = m_messagesInfo[port];

    if (message.isReliable())
    {
        // Set new counter ID
        message.setMessageID(peerInfo->m_lastSentReliableMessageID);
        ++peerInfo->m_lastSentReliableMessageID;
        // Copy message to buffer and prepare to re-sending
        // Create packet
        Packet* packet = new Packet();
        packet->m_packetID = m_newPacketID++;
        packet->initMessageData(message);
        packet->m_address = addr;
        packet->m_port = port;
        packet->m_messageID = message.getMessageID();

        m_allPackets.insert(packet->m_packetID, packet);
        peerInfo->m_reliableSendingPacketNumbers.push_back(packet->m_packetID);
        connect(packet, &Packet::signalPacketNeedToResending,
                this, &NetworkAPI::handlePacketNeedToResending);
        packet->waitAcknowledgement();
    }
    else
    {
        // Ignore acknowledgement message
        if (message.getType() != MessageType::RUDPAcknowledgement)
        {
            // Change counter
            message.setMessageID(peerInfo->m_lastSentUnreliableMessageID);
            ++peerInfo->m_lastSentUnreliableMessageID;
        }
    }
}

void NetworkAPI::startListen()
{
    m_socket.bind(QHostAddress::LocalHost, ServerPort);
    prepareToReadData();
}

void NetworkAPI::prepareToReadData()
```

```
{
    connect(&m_socket, &QUdpSocket::readyRead, this, &NetworkAPI::readReady);
}

void NetworkAPI::closeSocket()
{
    m_socket.close();
}

void NetworkAPI::sendUnicastMessage(quint16 port, NetMessage& message, const QHostAddress& address)
{
    if (m_isRUDPEnabled)
    {
        observeSendingMessage(message, address, port);
    }
    QByteArray clientData;
    SerializationMessageHelper::serializeMessageToByteArray(clientData, message);

    WriteData(clientData, address, port);
}

void NetworkAPI::readReady()
{
    QByteArray buffer(m_socket.pendingDatagramSize(), '\0');

    QHostAddress sender;
    quint16 port;
    m_socket.readDatagram(buffer.data(),buffer.size(), &sender, &port);

    std::unique_ptr<NetMessage> message = SerializationMessageHelper::deserializeMessageFromByteArray(buffer);

    if (m_isRUDPEnabled)
    {
        observeReceivedMessage(*message, sender, port);
    }
    else
    {
        // without rudp and acknoledgements
        processMessage(*message, sender, port);
    }
}

void NetworkAPI::handlePacketNeedToResending(quint64 packetID)
{
    if (m_allPackets.contains(packetID))
    {
        Packet* packet = m_allPackets[packetID];
        WriteData(packet->m_messageData,
                  packet->m_address,
                  packet->m_port);
        emit signalMessageForUI("Resend to " + QString::number(packet->m_port));
    }
}
```

ДОДАТОК В

Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++

Опис програмного модуля

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_ 18Б 13-1

Аркушів 10

Київ 2019

АНОТАЦІЯ

У додатку надана інформація про програмну частину симуляції фізичних об’єктів в клієнт-серверній системі.

Програмний застосунок розроблений мовою програмування C++ і фреймворком Qt. Для розробки симуляції було використано фізичний двигун Box2D. Було обрано динамічний підхід для зміни положення фізичних об’єктів та оптимальний варіант їх синхронізації, а саме синхронізація стану. Було реалізовано механізм синхронізації фізичних об’єктів, використовуючи протоколи транспортного рівня моделі OSI, та були порівнені результати їх роботи. Було реалізовано протокол RUDP, який вирішує такі проблеми UDP, як втрата пакетів і порядку, гарантування доставки.

Система є актуальною для симуляцій, які потребують встановлювання зв’язку між учасниками, проте використовують ненадійний протокол передачі даних.

ЗМІСТ

ЗАГАЛЬНІ ВІДОМОСТІ

Розроблена система забезпечує детермінованість, тобто фізичні характеристики предметів синхронізуються між клієнтами системи. Було обрано динамічний підхід для руху фізичних об’єктів та синхронізовано їх стан у симуляції. Уся мережева робота застосунку виконується шаром RUDP над UDP сокетом.

Експериментально було доведено, що механізми роботи протоколів TCP і UDP складають труднощі у клієнт-серверних застосунках. Використання TCP, а саме механізм, який гарантує надійну доставку пакетів, може спричиняти затримку, що іноді є надмірною для системи. Використання UDP не створює таких затримок, проте не встановлює зв’язку.

RUDP було реалізовано і використано для забезпечення надійної передачі даних між учасниками симуляції. RUDP забезпечує такі функції, як підтвердження доставки пакетів, повторна відправка втрачених пакетів також.

Багато існуючих клієнт-серверних застосунків мають необхідну для відправки інформацію, причому одна її частина може вимагати надійності відправки, а інша – ні. Проте використовується зазвичай один протокол у системі, що зумовлює вибір розробника у сторону ефективності чи надійності. Розроблюваного застосунку дозволяє самостійно вирішувати в яких випадках, йому необхідна гарантія доставки повідомлень, а в яких цим можна знехтувати. Гнучкість такого підходу, безперечно, є актуальним для сучасних клієнт-серверних симуляційних додатків.

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Головною задачею даного програмного продукту є забезпечення синхронної поведінки фізичних симуляційних об’єктів у клієнт-серверній системі.

Про функціонал розроблених симуляції та мережевого протоколу справедливо сказати наступне:

- симуляція є інтуїтивно зрозумілою для користування;
- організовано механізм колізій та зіштовхувань у симуляції;
- користувач має змогу створювати і управляти фізичним об’єктом;
- якщо користувач підключився до серверу, або самостійно створив симуляцію, стан його об’єкта буде синхронізовано між всіма існуючими учасниками.
- система є одноранговою, тому кожен учасник знає про існування інших учасників і синхронізує між ними стан свого об’єкта, якщо такий є;
- поведінка фізичних об’єктів для всіх учасників симуляції є синхронною та детермінованою;
- застосунок гарантує доставку, порядок і коректну обробку мережевих повідомлень у випадку втрати та дублікації пакетів;
- користувач системи може самостійно обирати пакети, доставку яких необхідно гарантувати.
- робота системи залишається надійною при досить великих навантаженнях на симуляцію, зокрема мережевих.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Система синхронної поведінки фізичних об’єктів складається з чотирьох внутрішніх підсистем і одного зовнішнього модуля. Головною підсистемою є головне вікно застосунку, який водночас є елементом компонування системи і графічного інтерфейсу. Зовнішнім модулем є Box2D, що використовується для реалізації руху та розрахунку колізій фізичних об’єктів. Модуль Симуляція інкапсулює в собі взаємодію з Box2D. Для забезпечення вводу користувача передбачено Шар вводу – модуль, який відповідає за поточний стан натиснення клавіш, не вдаючись у подробиці WinAPI реалізації. За роботу з сокетами, передачу та отримання пакетів і синхронізацію даних відповідає модуль Мережа.

Увесь код було детально документовано, поруч с сигнатурою методів та описом членів класу написано код, який несе інформацію про використання зазначених елементів. Тому хотілося б у цьому розділі розкрити концепцію застосування окремих підсистем розроблюваної симуляції.

Точкою входу є функція main() у файлі main.cpp. За допомогою екземпляра класу CmdParser оброблюємо параметри командного рядка. Тепер до усіх значень отриманих з аргументів, можна доступитися через цей сінглтон-об’єкт.

Клас MainWindow відповідає за графічний інтерфейс програми. Завдяки механізму сигналів і слотів, інші частини застосунку починають виконувати свою роботу, коли користувач натискає ту чи іншу кнопку меню. Цей клас підписується на сигнали синхронізації об’єктів, і коли вони спрацьовують, виконується метод, який власне і контролює відображення об’єктів на екран.

Було розроблено підсистему повідомлень для організації зручної комунікації між клієнтами системи. На вершині ієрархії лежить клас NetMessage. При передачі повідомлення мережею існує метод в NetworkAPI void **sendUnicastMessage**(quint16 port, NetMessage& message, const QHostAddress& address). Виходячи з сигнатури методу, можна зробити висновок,

що нас зовсім не цікавить, яке конкретно це повідомлення. Реалізовано за допомогою поліморфізму і наслідування – елементів ООП. Кожне повідомлення має свій тип, власний номер, може бути “надійним” або ні, тому ці характеристики зберігаються в класі NetMessage. Наступні функції задають інтерфейс серіалізації для усих нащадків NetMessage:

- virtual void ***serialize***(QDataStream& stream);
- virtual void ***deserialize***(QDataStream& stream);

Ці методи призначені для серіалізації та десеріалізації відповідно у і з stream. Іншими словами, конвертуємо наші конкретні повідомлення у бінарний вигляд та назад для передачі мережею. Класи-нащадки перевизначають ці методи з обов’язковим викликом базового методу в першу чергу.

Наперед невідомо, яке повідомлення необхідно буде серіалізувати, проте це і не потрібно.

Особливістю класу NetMessage є застосування у ньому породжувального патерну – Фабричний метод. За це відповідає наступна статична функція:
static std::unique_ptr<NetMessage> **createMessage**(MessageType type).

Підсистему Симуляція було реалізовано таким чином, щоб вона оновлювалася 60 разів у секунду за допомогою вбудованого у фреймворк Qt механізму сигналів і слотів. Слот runFrame() відповідає за оновлення кожного симуляційного кадру. Цей метод містить у собі інші приватні методи, за допомогою яких процес оновлення симуляції поділяється на конкретні логічні частини: preUpdate(), update(), render(), postUpdate().

Розбиття симуляції на такі елементи є дуже зручним для підтримки коду і системи загалом.

Основною підсистемою, яка відповідає за синхронізацію об’єктів у симуляції є підсистема Мережа. Зокрема, найбільшу цінність для системи грає клас NetworkAPI. Його слід роздивитися більш детально. NetworkAPI відповідає за роботу з QUDPSocket, всі системні повідомлення відправляються за допомогою даного класу. При реалізації NetworkAPI, використано породжуючий патерн програмування Singleton.

Нижче вказано опис до ключових методів цього класу, які мають особливе значення для розроблюваної системи:

- static NetworkAPI& getInstance() — повертає єдиний екземпляр класу;
- void init(bool isAuthoritarianClient, bool isRUDPEnabled = true) — якщо значення isAuthoritarianClient істине — ініціалізується серверна логіка, інакше — клієнтська. Параметр isRUDPEnabled встановлюватиме подальше використання RUDP у системі (значення за замовчуванням — true);
- void sendUnicastMessage(quint16 port, NetMessage& message, const QHostAddress& address = QHostAddress::LocalHost) – метод, який відправляє на певний порт і адресу передане повідомлення;
- void handlePacketNeedToResending(quint64 packetID) — приватний слот, який викликається, коли будь-який з пакетів вважається неотриманим, і необхідно його повторно відправити;
- void WriteData(QByteArray& data, const QHostAddress& address = QHostAddress::LocalHost, quint16 port = ServerPort) – виконує ту ж роботу, що й sendUnicastMessage(), проте на більш низькому рівні;
- void processMessage(NetMessage& message, const QHostAddress& addr, quint16 port) — делегування застосунку обробити повідомлення;
- void observeReceivedMessage(NetMessage& message, const QHostAddress& addr, quint16 port) — метод, який застосовується для RUDP, виконує усю необхідну логіку для забезпечення надійного отримання повідомлень;
- void observeSendingMessage(NetMessage& message, const QHostAddress& addr, quint16 port) — метод, який застосовується для RUDP, виконує усю необхідну логіку для забезпечення надійної відправки повідомлень;

У додатку Б подано реалізацію RUDP налаштування у цьому класі.

ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Програмний застосунок було розроблено за допомогою мови програмування C++ та фреймворку Qt. Було обрано стандарт мови C++11 тому, що він має потужний функціонал. В поданій системі були використанні: ключове слово auto, що дозволяє не вказувати тип змінної явно, говорячи компілятору, щоб він сам визначив фактичний тип змінної; строго-типізований enum, інтелектуальні показчики, які дозволяють безпечно виділяти пам’ять, не хвилюючись про її вивільнення та лямбда-функції.

Було використано механізм сигналів і слотів який нагадує систему подій. Слоти з’єднуються з сигналами і викликаються, коли випускається сигнал.

Робота з сокетом відбувається через клас QUdpSocket. Найпоширенішим способом використання цього класу є прив'язка до адреси та порту за допомогою методу bind(), потім виклику writeDatagram() і readDatagram() або receiveDatagram() для передачі даних.

Для реалізації фізичної симуляції я використовую Box2D. Box2D - це бібліотека для моделювання твердого тіла у двовимірному просторі для ігор. Об'єкти рухаються реалістично шляхом прикладання до них сил і роблять симуляцію більш інтерактивною. Box2D написано на C++.

Для налагодження системи вводу було використано метод GetAsyncKeyState(int) з заголовочного файлу windows.h для асинхронної перевірки, чи була натиснена певна клавіша.

Для обробки параметрів командного рядка я використовую клас QCommandLineParser. Він дозволяє у зручному режимі задати необхідні користувацькі параметри для командного рядка і опрацьовувати їх при запуску застосунку. Клас дозволяє перевіряти, чи були встановлені вказані аргументи і безпечно доступатися до їх значення.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Для того, щоб почати користуватися розробленою системою, необхідно розпакувати архів, в якому буде знаходитися виконуваний exe-файл і відповідні динамічні бібліотеки, необхідні для запуску. Робота фізичної симуляції з використанням Box2D є досить оптимізованою, тому для підтримання достатньої продуктивності буде достатньо процесора на базі Intel Core i3. Необхідною вимогою є підключення до мережі Інтернет.

Файли udp_demo.exe і tcp_demo.exe пропонують перевірити застосунок, використовуючи протоколи UDP і TCP відповідно. Файл udp_actual.exe відображає останню версію роботи програми, використовуючи UDP. Файл rudp.exe запускає симуляцію, використовуючи для мережевої частини RUDP з п’ятьма можливими спробами відправки пакета до отримувача. Файл rudp_8.exe запускає симуляцію, використовуючи для мережевої частини RUDP з вісьмома можливими спробами і є найоптимальнішим в контексті розроблюваної системи.

ВХІДНІ ТА ВИХІДНІ ДАНІ

Розроблений програмний застосунок очікує дій від користувача. За допомогою графічного інтерфейсу користувач може створити симуляцію або підключитися до вже існуючої, оновити застосунок, вийти з симуляції. Натиснувши праву кнопку миші, гравець створить об’єкт у вибраному місці. Після цього система буде реагувати на натиснені клавіші руху об’єкта і прикладати до нього силу в заданому напрямку. Тобто на вході ми маємо ввід користувача, на виході рух фізичного об’єкта. Поточний стан об’єкта синхронізується кожний кадр між усіма іншими учасниками симуляції.

Слід зазначити, що для клієнту не обов’язково створювати локально об’єкт. Він зможе спостерігати за іншими об’єктами, якщо буде авторитарним клієнтом, тобто створить симуляцію, або ж просто підключиться до існуючої симуляції.

ДОДАТОК Г

Реалізація синхронної поведінки фізичних об’єктів у клієнт-серверній системі з використанням C++

Копії публікацій

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51191_18Б 14-1

Аркушів 3

Київ 2019

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ

Матеріали XVII Міжнародної
науково-практичної конференції
молодих вчених та студентів
м. Київ, 23-26 квітня 2019 року,

ТОМ 2

Київ- 2019

Інтерактивна карта альтернативних джерел України.
ШИКЕР Б.Ю., студент гр. ТВ-51; БЕТІН В.С., студент гр. ТІ-51
Керівник - ст.викл., к.т.н. Матях С.В.

Конструктор гідроакустичних сигнатур морських об'єктів за характеристиками його компонентів.
СТЕПАНЮК С.О., студент гр. ТМ-51
Керівник - доц., к.т.н. Варава І.А.

Реалізація синхронної поведінки фізичних об'єктів у клієнт-серверній системі.
СКОРОБОГАТСЬКИЙ Д.В., студент гр. ТІ-51
Керівник - доц., к.т.н. Кузьменко І.М.

Розробка інтелектуального агенту для моніторингу та управління енергетичних потоків будівлі.
ПИРОГОВСЬКА Т.В., студент гр. ТІ-51
Керівник - доц., к.т.н. Ковальчук А.М.

Моделювання процесу обтікання крилового профілю на малих швидкостях.
ОНІСІМЧУК М.В., студент гр. ТР-51
Керівник - проф., д.ф.-м.н. Гуржій О.А.

Програмний агент моніторингу та управління сонячної електричної станції.
ЗАДАЧИН С.С., студент гр. ТМ-51
Керівник - доц., к.т.н. Ковальчук А.М.

Програмний агент моніторингу та управління вітроенергетичної установки.
ЗАДАЧИН Г.С., студент гр. ТМ-51
Керівник - доц., к.т.н. Ковальчук А.М.

Система порівняння гідроакустичних сигналів.
ДУДКО А.В., студент гр. ТР-51
Керівник - доц., к.т.н. Варава І.А.

Дослідження проблем тестування програмних продуктів.
ПАЦЕВКО О.О., студент гр. ТІ-72
Керівник - доц., к.т.н. Кублій Л.І.

Дослідження ефективності неявної різницевої схеми для рівняння теплопровідності .
КОЛОШМАЙ В.П., студент гр. ТР-71
Керівник - ст.викл. Молодід О.К.

Покращення точності розкладу функції в ряду Фур'є на проміжку [0;L].
КИБА І.А., студент гр. ТР-71
Керівник - ст.викл. Молодід О.К.

Дослідження ефективності явної різницевої схеми для рівняння теплопровідності.
ЖУРАВЛЬОВ Р.В., студент гр. ТР-71
Керівник - ст.викл. Молодід О.К.

Різноманітність застосування граничних теорем теорії ймовірностей на практиці .
ГОЛОВАЧУК С.В., студент гр. ТІ-72
Керівник - доц., к.т.н. Кублій Л.І.

Дослідження ефективності використання схеми Кранка-Ніколсона.
ВОЛКОВ О.В., студент гр. ТМ-72
Керівник - ст.викл. Молодід О.К.

УДК 608:2:004.032.24

Студент 4 курсу, гр. ТІ-51 Скоробогатський Д.В.
Доц., к.т.н. Кузьменко І.М.

РЕАЛІЗАЦІЯ СИНХРОННОЇ ПОВЕДІНКИ ФІЗИЧНИХ ОБ'ЄКТІВ У КЛІЄНТ-СЕРВЕРНІЙ СИСТЕМІ

На даний час важко переоцінити розвиток та потужність клієнт-серверних додатків. Це, зокрема, і соціальні мережі, месенджери, які дозволяють обмінюватися текстовими повідомленнями, мережа Інтернет загалом, комп'ютерні ігри. Перераховані застосунки потребують певної архітектурної технології, яка дозволить передавати дані між екземплярами програми найшвидше і з мінімізацією помилок.

Архітектура «клієнт-сервер» визначає загальні принципи взаємодії між комп'ютерами, правила визначають протоколи взаємодії, наприклад: HTTP, FTP, POP, SMTP, TELNET.

В даній роботі була поставлена задача збудувати клієнт-серверну систему для фізичної симуляції. Система має забезпечувати детермінованість, тому фізичні характеристики предметів: місце розміщення, швидкість руху, мають точно передаватися усім клієнтам системи. Для вирішення даної задачі працювали з транспортним рівнем моделі OSI (Open Systems Interconnection) з використанням сокетів [1]. Дані, що передавалися клієнтам, описували положення об'єктів системи в динаміці.

Широко відомим протоколом транспортного рівня є Transmission Control Protocol (TCP). Проте механізм роботи цього протоколу може складати певні труднощі у реалізації клієнт-серверних застосунків. У ході виконання системи було показано, що вже при 2 % втрати пакетів, що передаються, симуляція втрачає плавність руху об'єктів.

При використанні протоколу UDP відповідальність за обробку помилок і повторну передачу даних покладена на протоколи вищого рівня. Протокол UDP є ефективним для серверів, що надсилають невеликі відповіді великій кількості клієнтів. Але його недоліком для даної системи є значний час, що потрібен на опрацювання помилок при передачі даних.

У ході програмування зазначеної клієнт-серверної системи використано протокол, який поєднує в собі ефективність UDP і надійність TCP - Reliable Data Protocol (RDP) [2] . Його використовують для забезпечення надійної передачі даних між пакетно-орієнтованими застосунками. На відміну від UDP, RDP забезпечує такі функції, як підтвердження доставки пакетів, повторну відправка втрачених пакетів. Важливо зазначити, що клієнт застосунку може самостійно вирішувати у яких випадках, йому необхідна гарантія доставки його повідомлень.

Основою розроблюваної системи для тестування роботи протоколу стала симуляція фізичних об'єктів, яка базується на фізичному ігровому двигуні Box2D. Систему було розроблено у кросплатформному середовищі розробки Qt Creator з використанням мови програмування C++.

Перелік посилань:

1. State Synchronization. Keeping simulations in sync. [Електроннний ресурс] – Режим доступу: https://gafferongames.com/post/state_synchronization/
2. C. Partridge, R. Hinden: Specification RFC 1151 Experimental - Version 2 of the Reliable Data Protocol (RDP) (1990)